ÉTUDES DÉVELOPPEMENT

PRATIQUE DE MySQL ET PHP

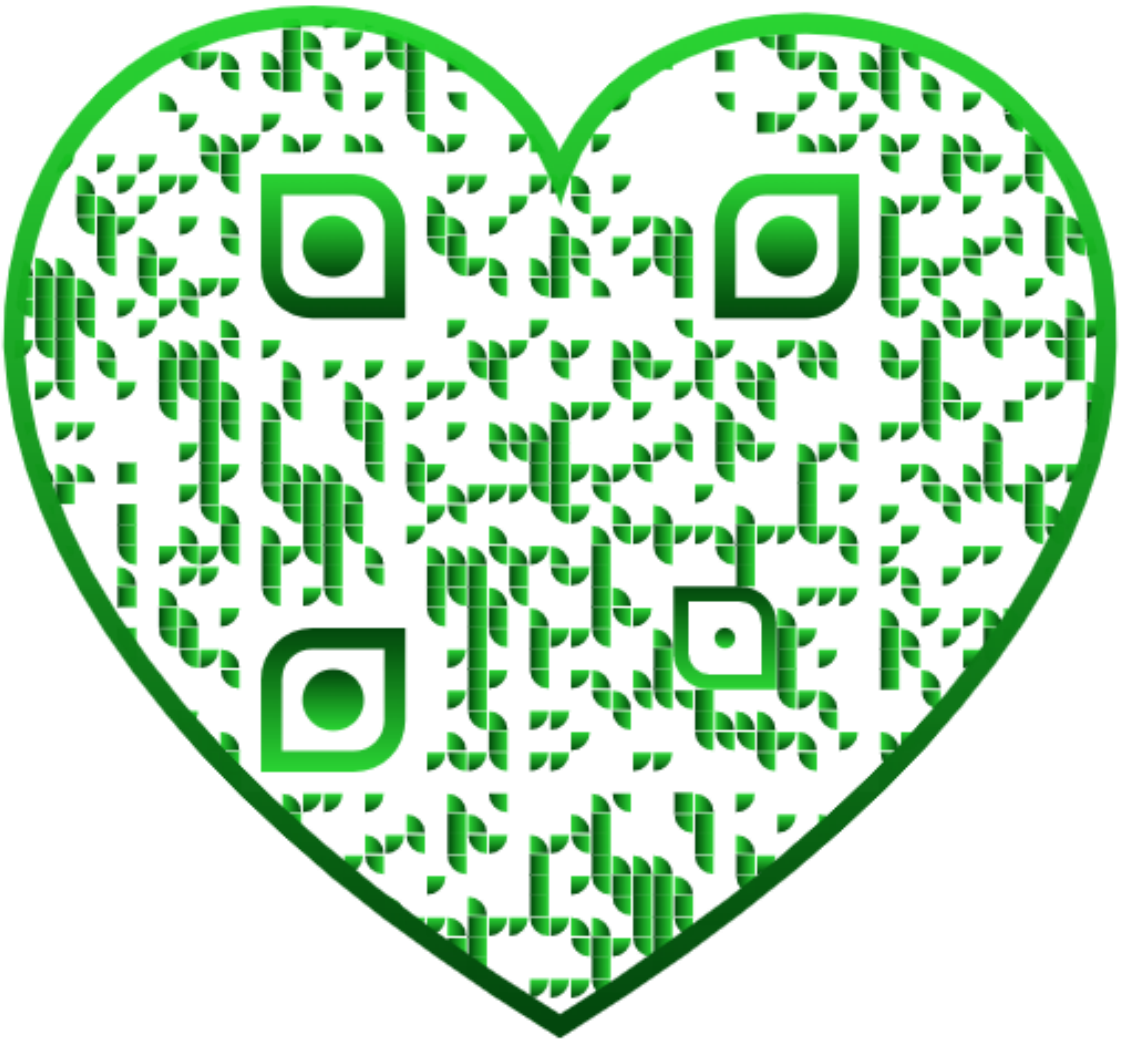
Conception et réalisation de sites web dynamiques

SUPPLÉMENTS en ligne

Philippe Rigaux

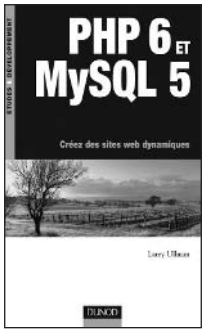
4e édition

DUNOD



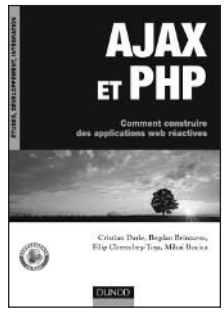
PRATIQUE DE MySQL ET PHP

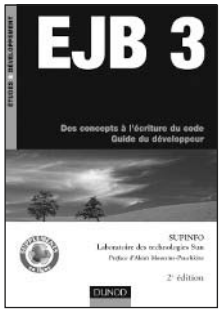
Consultez nos parutions sur dunod.com



PHP 6 et MySQL 5
Créez des sites web dynamiques
Larry Ullman
688 pages
Dunod, 2008

Ajax et PHP
Comment construire des applications web réactives
Christian Darie, Bogdan Brinzarea, Filip Chereches-Tosa, Mihai Bucica
312 pages
Dunod, 2007





EJB 3
Des concepts à l'écriture du code.
Guide du développeur
SUPINFO Laboratoire des technologies SUN
368 pages
Dunod, 2008

PRATIQUE DE MySQL ET PHP

Conception et réalisation de sites web dynamiques

Philippe Rigaux

Professeur des universités à Paris-Dauphine
et consultant en entreprises

4e édition

DUNOD

Toutes les marques citées dans cet ouvrage sont des marques déposées par leurs propriétaires respectifs.

Les trois premières éditions de cet ouvrage ont été publiées par O'Reilly France

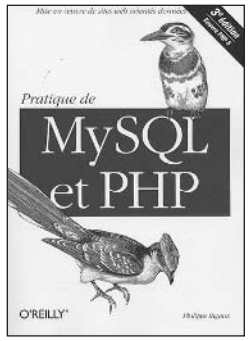


Illustration de couverture : Abejarucos © Juan Pablo Fuentes Serrano Fotolia.com

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

DANGER
LE PHOTOCOPILLAGE TUE LE LIVRE

© Dunod, Paris, 2009
ISBN 978-2-10-053752-5

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Table des matières

Avant-propos xv

Première partie – Programmation web avec MySQL/PHP

Chapitre 1 – Introduction à MySQL et PHP 3

1.1 Introduction au Web et à la programmation web 3

1.1.1 Serveurs web 4

1.1.2 Documents web : le langage XHTML 4

1.1.3 Programmation web 7

1.1.4 Sessions 16

1.2 Programmation web avec MySQL et PHP 18

1.2.1 MySQL 18

1.2.2 PHP 20

1.3 Une première base MySQL 24

1.3.1 Création d'une table 25

1.3.2 L'utilitaire mysql 25

1.3.3 L'interface PhpMyAdmin 34

1.4 Accès à MySQL avec PHP 36

1.4.1 L'interface MySQL/PHP 37

1.4.2 Formulaires d'interrogation 42

1.4.3 Formulaires de mises à jour 46

Chapitre 2 – Techniques de base 55

2.1 Programmation avec fonctions 56
2.1.1 Création de fonctions 56
2.1.2 Utilisation des fonctions 59
2.1.3 À propos de `require` *et* `include` 60
2.1.4 Passage par valeur et passage par référence 61

2.2 Traitement des données transmises par HTTP 64
2.2.1 Échappement et codage des données HTTP 67
2.2.2 Contrôle des données HTTP 70
2.2.3 Comment insérer dans la base de données : insertion dans MySQL 72
2.2.4 Traitement de la réponse 74
2.2.5 Comment obtenir du texte « pur » : envoi de l'e-mail 76
2.2.6 En résumé : traitement des requêtes et des réponses 77

2.3 Mise à jour d'une base par formulaire 78
2.3.1 Script d'insertion et de mise à jour 78
2.3.2 Validation des données et expressions régulières 86

2.4 Transfert et gestion de fichiers 90
2.4.1 Transfert du client au serveur 91
2.4.2 Transfert du serveur au client 95

2.5 Sessions 98
2.5.1 Comment gérer une session web ? 99
2.5.2 Gestion de session avec cookies 101
2.5.3 Prise en charge des sessions dans PHP 106

2.6 SQL dynamique et affichage multi-pages 109
2.6.1 Affichage d'une requête dynamique 110
2.6.2 Affichage multi-pages 111

Chapitre 3 – Programmation objet 115

3.1 Tour d'horizon de la programmation objet 116
3.1.1 Principes de la programmation objet 117
3.1.2 Objets et classes 120
3.1.3 Les exceptions 124
3.1.4 Spécialisation : classes et sous-classes 126
3.1.5 Spécialisation et classes abstraites : la classe `BD` 129

3.1.6 Résumé ... 138
3.2 La classe `Tableau` ... 140
3.2.1 Conception ... 140
3.2.2 Utilisation ... 144
3.2.3 Implantation ... 148
3.3 La classe `Formulaire` ... 152
3.3.1 Conception ... 152
3.3.2 Utilisation ... 154
3.3.3 Implantation ... 157
3.4 La classe `IhmBD` ... 167
3.4.1 Utilisation ... 168
3.4.2 Implantation ... 171

Deuxième partie – Conception et création d'un site

Chapitre 4 – Création d'une base MySQL ... 181
4.1 Conception de la base ... 181
4.1.1 Bons et mauvais schémas ... 181
4.1.2 Principes généraux ... 185
4.1.3 Création d'un schéma E/A ... 187
4.2 Schéma de la base de données ... 193
4.2.1 Transcription des entités ... 193
4.2.2 Associations de un à plusieurs ... 194
4.2.3 Associations de plusieurs à plusieurs ... 195
4.3 Création de la base *Films* avec MySQL ... 197
4.3.1 Tables ... 198
4.3.2 Contraintes ... 199
4.3.3 Modification du schéma ... 204

Chapitre 5 – Organisation du développement ... 207
5.1 Choix des outils ... 208
5.1.1 Environnement de développement intégré Eclipse ... 208
5.1.2 Développement en équipe : Subversion ... 210
5.1.3 Production d'une documentation avec PhpDoc ... 213

5.1.4 *Tests unitaires avec PhpUnit* 216
5.1.5 *En résumé* 220
5.2 Gestion des erreurs 221
5.2.1 *Erreurs syntaxiques* 222
5.2.2 *Gestion des erreurs en PHP* 225
5.2.3 *Les exceptions PHP* 227
5.2.4 *Gestionnaires d'erreurs et d'exceptions* 230
5.3 Portabilité multi-SGBD 233
5.3.1 *Précautions syntaxiques* 233
5.3.2 *Le problème des séquences* 235
5.3.3 *PDO, l'interface générique d'accès aux bases relationnelles* 238

Chapitre 6 – Architecture du site : le *pattern* MVC 241
6.1 Le motif de conception MVC 242
6.1.1 *Vue d'ensemble* 242
6.1.2 *Le modèle* 243
6.1.3 *La vue* 243
6.1.4 *Contrôleurs et actions* 243
6.1.5 *Organisation du code et conventions* 243
6.2 Structure d'une application MVC : contrôleurs et actions 245
6.2.1 *Le fichier index.php* 245
6.2.2 *Le contrôleur frontal* 248
6.2.3 *Créer des contrôleurs et des actions* 249
6.3 Structure d'une application MVC : la vue 251
6.3.1 *Les templates* 252
6.3.2 *Combiner des templates* 256
6.3.3 *Utilisation d'un moteur de templates comme vue* MVC 260
6.3.4 *Exemple complet* 261
6.3.5 *Discussion* 265
6.4 Structure d'une application MVC : le modèle 267
6.4.1 *Modèle et base de données : la classe* `TableBD` 267
6.4.2 *Un exemple complet de saisie et validation de données* 273
6.4.3 *Pour conclure* 277

Chapitre 7 – Production du site 279

7.1 Authentification 280

7.1.1 Problème et solutions 280

7.1.2 Contrôleur d'authentification et de gestion des sessions 281

7.1.3 Les actions de login et de logout 286

7.2 Recherche, présentation, notation des films 289

7.2.1 Outil de recherche et jointures SQL 289

7.2.2 Notation des films 295

7.3 Affichage des films et forum de discussion 299

7.4 Recommandations 304

7.4.1 Algorithmes de prédiction 305

7.4.2 Agrégation de données avec SQL 307

7.4.3 Recommandations de films 309

Chapitre 8 – XML 317

8.1 Introduction à XML 318

8.1.1 Pourquoi XML ? 319

8.1.2 XML et HTML 320

8.1.3 Syntaxe de XML 320

8.2 Export de données XML 323

8.2.1 Comment passer d'une base MySQL à XML 323

8.2.2 Application avec PHP 328

8.3 Import de données XML dans MySQL 332

8.3.1 SimpleXML 333

8.3.2 L'API SAX 335

8.3.3 Une classe de traitement de documents XML 339

8.4 Mise en forme de documents avec XSLT 348

8.4.1 Quelques mots sur XSLT 349

8.4.2 Application d'un programme XSLT avec PHP 353

Troisième partie – Compléments

Chapitre 9 – Introduction au *Zend Framework* 357

9.1 Mise en route 358
9.1.1 Installation d'une application ZF 358
9.1.2 Redirection des requêtes avec le ZF 359
9.1.3 Organisation et conventions 360
9.1.4 Routage des requêtes dans une application Zend 362
9.1.5 Configuration 365
9.1.6 Connexion à la base de données 366
9.1.7 Le registre 367
9.1.8 Contrôleurs, actions et vues 367

9.2 Accès à la base de données 369
9.2.1 Interrogation 370
9.2.2 Insertion et mise à jour 372

9.3 Le MVC du Zend Framework 373
9.3.1 L'objet `request` 373
9.3.2 L'objet `response` 374
9.3.3 Gérer les exceptions 374

9.4 La vue dans le Zend Framework 376
9.4.1 Les vues sont des scripts PHP 376
9.4.2 Le layout 377
9.4.3 Créer des Helpers 378

9.5 Le composant *Modèle* du Zend Framework 379
9.5.1 L'ORM du Zend Framework 379
9.5.2 Le modèle ORM de l'application 380
9.5.3 Manipulation des données avec les classes ORM 383

9.6 Pour conclure 385

Chapitre 10 – Récapitulatif SQL 387

10.1 Sélections 388
10.1.1 Renommage, fonctions et constantes 389
10.1.2 La clause `DISTINCT` 392
10.1.3 La clause `ORDER BY` 393

10.1.4 La clause `WHERE` ... 393
10.1.5 Dates ... 396
10.1.6 Valeurs nulles ... 396
10.1.7 Clauses spécifiques à MySQL ... 398
10.2 Jointures ... 399
10.2.1 Interprétation d'une jointure ... 399
10.2.2 Gestion des ambiguïtés ... 401
10.2.3 Jointures externes ... 404
10.3 Opérations ensemblistes ... 405
10.4 Requêtes imbriquées ... 406
10.4.1 Exemples de requêtes imbriquées ... 407
10.4.2 Requêtes corrélées ... 409
10.4.3 Requêtes avec négation ... 411
10.5 Agrégation ... 413
10.5.1 La clause `GROUP BY` ... 413
10.5.2 La clause `HAVING` ... 415
10.6 Mises à jour ... 416
10.6.1 Insertion ... 416
10.6.2 Destruction ... 417
10.6.3 Modification ... 417

Chapitre 11 – Récapitulatif PHP ... 419
11.1 Généralités ... 419
11.1.1 Commentaires ... 420
11.1.2 Variables et littéraux ... 420
11.1.3 Constantes ... 421
11.2 Types ... 422
11.2.1 Types numériques et booléens ... 422
11.2.2 Chaînes de caractères ... 422
11.2.3 Tableaux ... 423
11.2.4 Conversion et typage ... 425
11.3 Expressions ... 426

11.4 Opérateurs 427
11.4.1 Concaténation de chaînes 428
11.4.2 Incrémentations 428
11.4.3 Opérateurs de bits 429
11.4.4 Opérateurs logiques 429
11.5 Structures de contrôle 430
11.5.1 Tests 431
11.5.2 Boucles 432
11.5.3 Les instructions `break` *et* `continue` 434
11.6 Fonctions 435
11.6.1 Passage des arguments 435
11.6.2 Valeurs par défaut 437
11.6.3 Fonctions et variables 437
11.7 Programmation orientée-objet 440
11.7.1 Classes et objets 440
11.7.2 Constructeurs et destructeurs 441
11.7.3 Sous-classes 442
11.7.4 Manipulation des objets 442
11.7.5 Compléments 443

Quatrième partie – Annexes

Annexe A – Installation Apache/PHP/MySQL 447
A.1 Mot de passe `root` 447
A.2 Création de bases et d'utilisateurs 448
A.2.1 La commande `GRANT` 449
A.2.2 Modification des droits d'accès 451
A.3 Fichiers de configuration 452
A.3.1 Format d'un fichier de configuration 452
A.3.2 Les différents fichiers 452
A.3.3 Configuration du serveur 453
A.3.4 Configuration d'un compte administrateur 454
A.4 Sauvegardes 455
A.5 phpMyAdmin 457

Annexe B – Référence MySQL 461
B.1 Types de données MySQL 461
B.2 Commandes de MySQL 465
B.3 Fonctions MySQL 475

Annexe C – Fonctions PHP 485
C.1 Fonctions générales 486
C.2 Chaînes de caractères 493
C.3 Dates 496
C.4 Tableaux 497
C.5 Fonctions XML 500
C.6 Accès aux fichiers 504
C.7 Interface PHP/MySQL 507

Index général 517

Index des fonctions PHP 523

Index des commandes SQL 527

Table des figures 531

Avant-propos

Quand la première édition de ce livre est parue, en janvier 2001, la réputation de MySQL et de PHP était déjà bien établie. Ces deux outils étaient connus pour être fiables, performants, pratiques et bien adaptés à une utilisation très spécialisée : la production dynamique de pages HTML. En revanche, pris isolément et dans un contexte plus général de développement d'applications bases de données, ni MySQL ni PHP ne semblaient en mesure de rivaliser avec des logiciels commerciaux nettement plus puissants et complets.

Huit ans après cette première édition tout ou presque a changé. MySQL est un SGBD reconnu, doté de toutes les fonctionnalités requises pour un système relationnel. La version 5 (et bientôt la version 6) de PHP est maintenant bien installée et constitue un langage de programmation abouti que ses concepteurs et développeurs se sont acharnés à améliorer pour le placer au même niveau que Java ou le C++. De plus la maturité de ces deux outils a favorisé la parution d'environnements de développement avancés, incluant tous les outils d'ingénierie logicielle (éditeurs intégrés, production de documentation, bibliothèques de fonctionnalités prêtes à l'emploi, débogueurs, etc.) qui les rendent aptes à la production de logiciels répondant à des normes de qualités professionnelles. Même pour des projets d'entreprise importants (plusieurs années-homme), l'association MySQL/PHP est devenue tout à fait compétitive par rapport à d'autres solutions parfois bien plus lourdes à concevoir, mettre en place et entretenir.

Objectifs et contenu de ce livre

Ce livre présente l'utilisation de MySQL et de PHP pour la production et l'exploitation de sites web s'appuyant sur une base de données. Son principal objectif est d'exposer de la manière la plus claire et la plus précise possible les techniques utilisées pour la création de sites web interactifs avec MySQL/PHP. Il peut s'énoncer simplement ainsi :

> Donner au lecteur la capacité à résoudre lui-même tous les problèmes rencontrés dans ce type de développement, quelle que soit leur nature ou leur difficulté.

Ce livre n'énumère pas toutes les fonctions PHP : il en existe des milliers et on les trouve très facilement dans la documentation en ligne sur *http://www.php.net*, toujours plus complète et à jour que n'importe quel ouvrage. Je ne donne pas non

plus, sauf pour quelques exceptions, une liste de ressources sur le Web. Elles évoluent rapidement, et celles qui existent se trouvent de toute manière sans problème. En revanche, le livre vise à expliquer le plus clairement possible *comment* et *pourquoi* on en arrive à utiliser telle ou telle technique, et tente de donner au lecteur les moyens d'intégrer ces connaissances pour pouvoir les réutiliser le moment venu dans ses propres développements.

La lecture de chaque chapitre doit permettre de progresser un peu plus dans la compréhension de la nature des problèmes (comment puis-je le résoudre ?) et de la méthode menant aux solutions (pourquoi PHP et MySQL sont-ils intéressants pour y arriver concisément et élégamment ?). Les concepts et outils sont donc présentés d'une manière *pratique*, *progressive* et *complète*.

- **Pratique**. Ce livre ne contient pas d'exposé théorique qui ne soit justifié par une application pratique immédiate. Les connaissances nécessaires sur la programmation PHP, le langage SQL ou les bases de données relationnelles sont introduites au fur et à mesure des besoins de l'exposé.
- **Progressive**. L'ordre de la présentation est conçu de manière à donner le plus rapidement possible des éléments concrets pour expérimenter toutes les techniques décrites. Le livre adopte ensuite une démarche très simple consistant à détailler depuis le début les étapes de construction d'un site basé sur MySQL et PHP, en fournissant un code clair, concis et aisément adaptable.
- **Complète**. Idéalement, un seul livre contiendrait toutes les informations nécessaires à la compréhension d'un domaine donné. C'est utopique en ce qui concerne les techniques de gestion de sites web. J'ai cherché en revanche à être aussi complet que possible pour tout ce qui touche de près à la programmation en PHP d'applications web s'appuyant sur une base de données.

Ce livre vise par ailleurs à promouvoir la *qualité* technique de la conception et du développement, qui permettra d'obtenir des sites maintenables et évolutifs. MySQL et PHP sont des outils relativement faciles à utiliser, avec lesquels on obtient rapidement des résultats flatteurs, voire spectaculaires. Cela étant, il est bien connu qu'il est plus facile de développer un logiciel que de le faire évoluer. Une réalisation bâclée, si elle peut faire illusion dans un premier temps, débouche rapidement sur des problèmes récurrents dès qu'on entre en phase de maintenance et d'exploitation.

Une seconde ambition, complémentaire de celle mentionnée précédemment, est donc d'introduire progressivement tout au long du livre des réflexions et des exemples qui montrent comment on peut arriver à produire des logiciels de plus en plus complexes en maîtrisant la conception, le développement et l'enrichissement permanent de leurs fonctionnalités. Cette maîtrise peut être obtenue par des techniques de programmation, par la mise en œuvre de concepts d'ingénierie logicielle éprouvés depuis de longues années et enfin par le recours à des environnements de programmation avancés (les « *frameworks* »). Cette seconde ambition peut se résumer ainsi :

Montrer progressivement au lecteur comment on passe de la réalisation de sites dynamiques légers à des applications professionnelles soumises à des exigences fortes de qualité.

Cette quatrième édition reprend pour l'essentiel Le contenu de la précédente, avec des modifications visant, quand cela m'a semblé possible, à améliorer la clarté et la simplicité des exemples et des explications. J'ai ajouté un chapitre dédié aux environnements de programmation PHP/MySQL, en introduisant notamment le *Zend Framework* pour illustrer l'aboutissement actuel d'une démarche de normalisation de la production d'applications web basée sur des concepts qui ont fait leurs preuves. Ce chapitre vient bien sûr en fin d'ouvrage car il intéressera surtout ceux qui visent à réaliser des applications d'envergure. Les autres pourront se satisfaire des techniques présentées dans les chapitres précédents, qui demandent un apprentissage initial moins ardu. Un souci constant quand on écrit ce genre d'ouvrage est de satisfaire le plus grand nombre de lecteurs, quels que soient leurs connaissances de départ ou leurs centres d'intérêt. J'espère que les évolutions apportées et la réorganisation du livre atteindront cet objectif.

Audience et pré-requis

Ce livre est en principe auto-suffisant pour ce qui concerne son sujet : la programmation d'applications web dynamiques avec MySQL et PHP. Il est clair qu'au départ une compréhension des notions informatiques de base (qu'est-ce qu'un réseau, un fichier, un éditeur de texte, un langage de programmation, une compilation, etc.) est préférable. Je suppose également connues quelques notions préalables comme la nature de l'Internet et du Web, les notions de client web (navigateur) et de serveur web, et les bases du langage HTML. On trouve très facilement des tutoriaux en ligne sur ces sujets.

Les techniques de programmation PHP, des plus simples (mise en forme HTML de données) aux plus complexes (divers exemples de programmation orientée-objet) sont introduites et soigneusement expliquées au fur et à mesure de la progression des chapitres. Le livre comprend également des exposés complets sur la conception de bases de données relationnelles et le langage SQL. Aucune connaissance préalable sur les bases de données n'est ici requise.

Un site web complète ce livre, avec des exemples, le code de l'application WEBSCOPE dont je décris pas à pas la réalisation, des liens utiles et des compléments de documentation. Voici son adresse :

http://www.lamsade.dauphine.fr/rigaux/mysqlphp

Le « WEBSCOPE » dont le développement est traité de manière complète dans la seconde partie est une application de « filtrage coopératif » consacrée au cinéma. Elle propose des outils pour rechercher des films, récents ou classiques, consulter leur fiche, leur résumé, leur affiche. L'internaute peut également noter ces films en leur attribuant une note de 1 à 5. À terme, le site dispose d'un ensemble suffisant d'informations sur les goûts de cet internaute pour lui proposer des films susceptibles de lui plaire. Vous pouvez vous faire une idée plus précise de ce site en le visitant et en l'utilisant, à l'adresse suivante :

http://www.lamsade.dauphine.fr/rigaux/webscope

Une fois le code récupéré, vous pouvez bien sûr l'utiliser, le consulter ou le modifier pour vos propres besoins. Par ailleurs, si vous souhaitez vous initier aux techniques de développement en groupe, j'ai ouvert sur *SourceForge.net* un projet WEBSCOPE consacré à ce site. Vous pourrez participer, avec d'autres lecteurs, à l'amélioration du code ainsi qu'à son extension, et apprendre à utiliser des outils logiciels avancés pour le développement de logiciels *Open Source*. Le site de ce projet est

http://webscope.sourceforge.net

Comment apprendre à partir du livre

À l'issue de la lecture de ce livre, vous devriez maîtriser suffisamment l'ensemble des concepts et outils nécessaires aux applications MySQL/PHP pour être autonome (ce qui n'exclut pas, au contraire, le recours aux diverses ressources et références disponibles sur le Web). L'acquisition de cette maîtrise suppose bien entendu une implication personnnelle qui peut s'appuyer sur les éléments suivants :

1. la lecture attentive des explications données dans le texte ;
2. l'utilisation des exemples fournis, leur étude et leur modification partielle ;
3. des exercices à réaliser vous-mêmes.

Pour tirer le meilleur parti des exemples donnés, il est souhaitable que vous disposiez dès le début d'un environnement de travail. Voici quelques recommandations pour mettre en place cet environnement en moins d'une heure.

Le serveur de données et le serveur web

Vous devez disposer d'un ordinateur équipé de MySQL et PHP. Il s'agit d'un environnement tellement standard qu'on trouve des *packages* d'installation partout. Les environnements Windows disposent de *Xampp*[1] et Mac OS de *MAMP*[2]. Ces logiciels se téléchargent et s'installent en quelques clics. Sous Linux ce n'est pas nécessairement plus compliqué, mais l'installation peut dépendre de votre configuration. Une recherche « *LAMP* » sur le Web vous donnera des procédures d'installation rapide.

La configuration

Normalement, les systèmes AMP (Apache-MySQL-PHP) sont configurés correctement et vous pouvez vous contenter de cette configuration par défaut au début. Un aspect parfois sensible est le fichier *php.ini* qui contient l'ensemble des paramètres fixant la configuration de PHP. La première chose à faire est de savoir où se trouve ce fichier. C'est assez simple : placez dans le répertoire *htdocs* de votre installation un fichier *phpinfo.php* avec le code suivant :

```
<?php phpinfo(); ?>
```

1. *http://www.apachefriends.org/en/xampp.html*
2. *http://www.mamp.info/en/index.php*

Accédez ensuite à l'URL *http://localhost/phpinfo.php*. Entre autres informations vous verrez l'emplacement de *php.ini* et tous les paramètres de configuration. Profitez de l'occasion pour vérifier les paramètres suivants (même si vous ne les comprenez pas pour l'instant) :

1. `short_open_tags` doit être à `Off` pour interdire d'utiliser des balises PHP abrégées ;
2. `display_errors` devrait valoir `On` ;
3. `file_upload` devrait valoir `On`.

Cela devrait suffire pour pouvoir débuter sans avoir de souci.

Le navigateur

Vous devez utiliser un navigateur web pour tester les exemples et le site. Je vous recommande fortement Firefox, qui présente l'avantage de pouvoir intégrer des modules très utiles (les *plugins*) qui le personnalisent et l'adaptent à des tâches particulières. L'extension *Web Developer* est particulièrement intéressante pour les développements web car elle permet de contrôler tous les composants transmis par le serveur d'une application web dynamique au navigateur.

Figure 1 — Barre d'outils Web Developer

La figure 1 montre l'intégration de Web Developer à Firefox sous la forme d'une barre d'outils offrant des possibilités d'inspection et de manipulation des composants CSS, JavaScript, *cookies*, etc. Ces possibilités s'avèrent extrêmement utiles pour la vérification des composants clients. Dans la mesure où il ne s'agit pas de notre préoccupation principale pour ce livre, je ne détaille pas plus les possibilités offertes.

Une fonction très simple et très utile est la validation du code HTML fourni au navigateur. La figure montre, sur la droite de la barre d'outils, des indicateurs

qui signalent un éventuel problème de conformité aux normes, etc. Ces indicateurs devraient toujours être au vert. Tout le code HTML décrit dans ce livre est conforme aux normes, et je vous conseille d'adopter dès le début cette bonne habitude.

L'extension s'installe comme toutes les autres dans Firefox, en passant par le menu *Outils*, *Modules complémentaires*.

L'environnement de développement

Un simple éditeur de texte suffit pour modifier les exemples et créer vos propres scripts. Essayez de trouver quand même mieux que le bloc-note de Windows. Des logiciels comme EditPlus ou UltraEdit font parfaitement l'affaire. Si vous souhaitez un outil plus avancé (mais plus difficile à manier pour les débutants) je vous recommande bien entendu Eclipse (*http://www.eclipse.org*) avec l'utilisation d'une perspective PHP. Le chapitre 5 présente brièvement cet environnement de développement intégré (IDE).

Exercices et exemples

Tous les exemples fournis, y compris le site complet dont la réalisation est intégralement décrite, sont conçus pour répondre aux trois contraintes suivantes :

1. ils sont testés et fonctionnent ;
2. ils sont corrects, autrement dit chaque fragment de code donné en exemple a un objectif bien identifié, et remplit cet objectif ;
3. ils visent, autant que possible, à rester clairs et concis.

Ces contraintes, parfois difficiles à satisfaire, contribuent à montrer que l'on peut développer des fonctionnalités parfois complexes en conservant un code accessible et maîtrisable. Un avantage annexe, quoique appréciable, est de vous permettre facilement d'obtenir, à partir d'un exemple qui tourne, une base de travail pour faire vos propres modifications et expérimentations.

Allez sur le site du livre et récupérez le fichier *exemples.zip*. Placez-le dans le répertoire *htdocs* de votre environnement MySQL/PHP et extrayez les fichiers. Si les serveurs sont démarrés, vous devriez pouvoir accéder à l'URL

htpp://localhost/exemples

et vous avez tous les exemples du livre (à l'exception de ceux intégrés au site WEBSCOPE) sous la main pour travailler parallèlement à votre lecture.

Organisation

Ce livre comprend trois parties et des annexes.

- La première partie est une présentation détaillée de toutes les techniques de base intervenant dans la construction de pages web basées sur MySQL et PHP : bases de la programmation web, création de tables MySQL, création de scripts PHP, accès à MySQL avec PHP, etc.

Cette partie comprend un chapitre qui explique comment réaliser les fonctions les plus courantes d'un site web dynamique : découpage d'un script en fonctions, gestion de formulaires HTML, transfert et gestion de fichiers, sessions et traitement des erreurs. Ces fonctions sont expliquées indépendamment d'une application particulière.
Le dernier chapitre de cette partie est entièrement consacré à la programmation orientée-objet, et montre comment concevoir des modules (ou *classes*) qui facilitent ensuite considérablement les tâches répétitives et routinières pendant le développement d'un site.

- La deuxième partie est consacrée à la conception et à la réalisation complète d'un site web, comprenant la conception de la base, l'organisation du code et la méthode de développement, l'authentification des utilisateurs et la production du site. Outre la génération, classique, des pages HTML, des chapitres sont consacrés à l'utilisation de XML pour l'échange et la publication de données, et à la production dynamique de graphiques.
- La troisième partie propose une introduction à un environnement de développement avancé (le *Zend Framework*) un récapitulatif du langage SQL, déjà présenté de manière progressive dans les deux premières parties, et un récapitulatif du langage PHP.

Un ensemble d'annexes donnant en ordre alphabétique les principales commandes, options et utilitaires de MySQL et de PHP, ainsi que quelques conseils d'administration, conclut le livre.

Conventions

J'utilise les conventions typographiques suivantes :

- La police `à chasse constante` s'applique à tous les exemples de code, de commandes et de programmes, que ce soit un *shell* UNIX, SQL, PHP, etc.
- La police *`à chasse constante en italiques`* est utilisée pour distinguer les paramètres des mots-clés dans la syntaxe des commandes.
- Le texte en *italiques* est utilisé pour les URL, les noms de fichiers, de programmes et de répertoires cités dans le texte (autrement dit, non inclus dans du code). L'italique est également utilisé pour les termes étrangers et pour la mise en valeur de mots ou d'expressions importants.

De plus, le code s'appuie sur des conventions précises pour nommer les fichiers, les variables, les fonctions, les noms de tables, etc. Ces conventions font partie d'une stratégie générale de qualité du développement et seront présentées le moment venu.

Remerciements

Je souhaite remercier chaleureusement tous ceux qui sont à l'origine de ce livre, ont permis sa réalisation ou contribué à l'amélioration du manuscrit. Merci donc à Bernd Amann, Joël Berthelin, Olivier Boissin, Bertrand Cocagne, Cécile, Hugues et Manuel Davy, Jean-François Diart, Cédric du Mouza, David Gross, Cyrille Guyot,

Alain Maës, Joël Patrick, Michel Scholl, François-Yves Villemin, Dan Vodislav, Emmanuel Waller et aux nombreux lecteurs qui m'ont suggéré des améliorations.

J'ai également bénéficié des remarques et des conseils de personnes auxquelles je tiens à exprimer plus particulièrement ma reconnaissance : Robin Maltête avec qui j'ai réalisé de nombreux sites et qui m'a apporté de nombreux problèmes stimulants à résoudre ; Michel Zam pour des discussions très instructives sur la conception et la réalisation de logiciel robustes et élégants ; Xavier Cazin qui a été à l'origine de ce livre et à qui je dois de très nombreuses et utiles remarques sur son contenu. Enfin, merci à Jean-Luc Blanc qui m'a accordé sa confiance et son temps pour la réalisation de cette quatrième édition.

PREMIÈRE PARTIE

Programmation web avec MySQL/PHP

1

Introduction à MySQL et PHP

Ce chapitre est une introduction à l'association de MySQL et de PHP pour la production de documents « dynamiques », autrement dit créés à la demande par un programme. Nous commençons par une courte introduction au web et à la programmation web, limitée aux pré-requis indespensables pour comprendre la suite du livre. Les lecteurs déjà familiers avec ces bases peuvent sauter sans dommage la section 1.1.

1.1 INTRODUCTION AU WEB ET À LA PROGRAMMATION WEB

Le *World-Wide Web* (ou WWW, ou Web) est un grand – très grand – système d'information réparti sur un ensemble de *sites* connectés par le réseau Internet. Ce système est essentiellement constitué de *documents hypertextes*, ce terme pouvant être pris au sens large : textes, images, sons, vidéos, etc. Chaque site propose un ensemble plus ou moins important de documents, transmis sur le réseau par l'intermédiaire d'un *programme serveur*. Ce programme serveur dialogue avec un *programme client* qui peut être situé n'importe où sur le réseau. Le programme client prend le plus souvent la forme d'un *navigateur*, grâce auquel un utilisateur du Web peut demander et consulter très simplement des documents. Le Web propose aussi des services ou des modes de communication entre machines permettant d'effectuer des calculs répartis ou des échanges d'information sans faire intervenir d'utilisateur : le présent livre n'aborde pas ces aspects.

Le dialogue entre un programme serveur et un programme client s'effectue selon des règles précises qui constituent un *protocole*. Le protocole du Web est HTTP, mais il est souvent possible de communiquer avec un site *via* d'autres protocoles, comme

par exemple FTP qui permet d'échanger des fichiers. Ce livre n'entre pas dans le détail des protocoles, même si nous aurons occasionnellement à évoquer certains aspects de HTTP.

1.1.1 Serveurs web

Un site est constitué, matériellement, d'un ordinateur connecté à l'Internet, et d'un programme tournant en permanence sur cet ordinateur, le *serveur*. Le programme serveur est en attente de requêtes transmises à son attention sur le réseau par un programme client. Quand une requête est reçue, le programme serveur l'analyse afin de déterminer quel est le document demandé, recherche ce document et le transmet au programme client. Un autre type important d'interaction consiste pour le programme client à demander au programme serveur d'exécuter un programme, en fonction de paramètres, et de lui transmettre le résultat.

La figure 1.1 illustre les aspects essentiels d'une communication web pour l'accès à un document. Elle s'effectue entre deux programmes. La requête envoyée par le programme client est reçue par le programme serveur. Ce programme se charge de rechercher le document demandé parmi l'ensemble des fichiers auxquels il a accès, et transmet ce document.

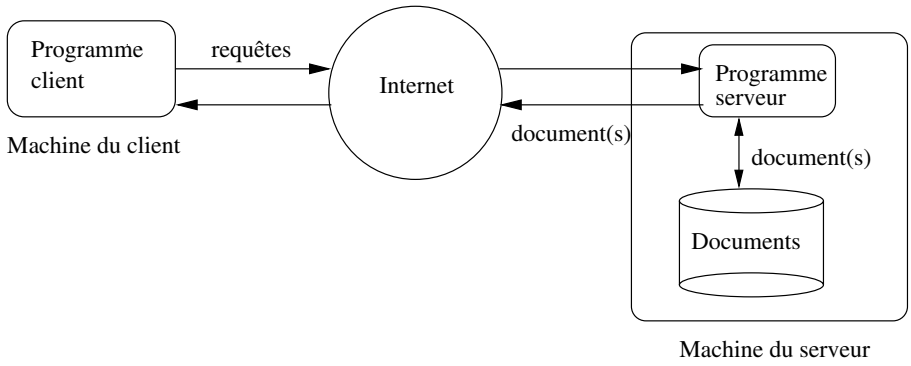


Figure 1.1 — Architecture web

Dans tout ce qui suit, le programme serveur sera simplement désigné par le terme « serveur » ou par le nom du programme particulier que nous utilisons, Apache. Les termes « navigateur » et « client » désigneront tous deux le programme client (Firefox, Safari, Internet Explorer, etc.). Enfin, le terme « utilisateur » (ou, parfois, « internaute »), sera réservé à la personne physique qui utilise un programme client.

1.1.2 Documents web : le langage XHTML

Les documents échangés sur le Web peuvent être de types très divers. Le principal est le *document hypertexte*, un texte dans lequel certains mots, ou groupes de mots, sont des *liens*, ou *ancres*, donnant accès à d'autres documents. Le langage qui permet de

spécifier des documents hypertextes, et donc de fait le principal langage du Web, est HTML.

La présentation de HTML dépasse le cadre de ce livre. Il existe de très nombreux tutoriaux sur le Web qui décrivent le langage (y compris XHTML, la variante utilisée ici). Il faut noter que HTML est dédié à la *présentation* des documents d'un site, et ne constitue pas un langage de programmation. Son apprentissage (au moins pour un usage simple) est relativement facile. Par ailleurs, il existe de nombreux éditeurs de documents HTML – on peut citer *DreamWeaver* ou *FrontPage* sous Windows, *Quanta+* sous Linux – qui facilitent le travail de saisie des balises et fournissent une aide au positionnement (ou plus exactement au pré-positionnement puisque c'est le navigateur qui sera en charge de la mise en forme finale) des différentes parties du document (images, menus, textes, etc.). Notre objectif dans ce livre n'étant pas d'aborder les problèmes de mise en page et de conception graphique de sites web, nous nous limiterons à des documents HTML relativement simples, en mettant l'accent sur leur intégration avec PHP et MySQL.

Voici un exemple simple du type de document HTML que nous allons créer. Notez les indications en début de fichier (`DOCTYPE`) qui déclarent un contenuse conformant à la norme XHTML.

Exemple 1.1 *exemples/ExHTML2.html* : *Un exemple de document XHTML*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Pratique de MySQL et PHP</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>
Pratique de <a href="http://www.mysql.com">MySQL</a>
et <a href="http://www.php.net">PHP</a>
</h1>

<hr/>

Ce livre, publié aux
<a href="http://www.dunod.fr">Éditions <i>Dunod</i></a>,
est consacré aux techniques
de création de sites à l'aide du langage
<a href="http://www.php.net"><b>PHP</b></a> et
du serveur  <a href="http://www.mysql.com"><b>MySQL</b></a>.

</body>
</html>
```

XHTML, une variante de HTML.

La variante utilisée dans nos exemples est XHTML, une déclinaison de HTML conforme aux règles de constitution des documents XML, un autre langage de description que nous étudierons dans le chapitre 8. XHTML impose des contraintes rigoureuses, ce qui représente un atout pour créer des sites portables et multi-navigateurs. Je vous conseille d'équiper votre navigateur d'un validateur HTML (comme *Web Developer* pour Firefox) qui vous indiquera si vos pages sont bien formées.

Codage des documents.

Nos documents sont encodés en ISO 8859 1 ou Latin1, un codage adapté aux langues européennes et facile à manipuler avec tous les éditeurs de texte disponibles sur les ordinateurs vendus en France. Si vous développez un site en multi-langues, il est recommandé d'adopter le codage UTF-8, qui permet par exemple de représenter des langues asiatiques. Il faut alors utiliser un éditeur qui permet de sauvegarder les documents dans ce codage.

Référencement des documents

Un des principaux mécanismes du Web est le principe de localisation, dit *Universal Resource Location* (URL), qui permet de faire référence de manière unique à un document. Une URL est constituée de plusieurs parties :

- le nom du protocole utilisé pour accéder à la ressource ;
- le nom du serveur hébergeant la ressource ;
- le numéro du port réseau sur lequel le serveur est à l'écoute ;
- le chemin d'accès, sur la machine serveur, à la ressource.

À titre d'exemple, voici l'URL du site web de ce livre :

http://www.lamsade.dauphine.fr/rigaux/mysqlphp/index.html

Cette URL s'interprête de la manière suivante : il s'agit d'un document accessible *via* le protocole HTTP, sur le serveur *www.lamsade.dauphine.fr* qui est à l'écoute sur le port 80 – numéro par défaut, donc non précisé dans l'URL – et dont le chemin d'accès et le nom sont respectivement *rigaux/mysqlphp* et *index.html*.

La balise qui permet d'insérer des liens dans un document HTML est le conteneur <**a**> pour *anchor* – « ancre » en français. L'URL qui désigne le lien est un *attribut* de la balise. Voici par exemple comment on associe l'expression « pratique de MySQL et PHP » à son URL.

```
<a href="http://www.lamsade.dauphine.fr/rigaux/mysqlphp">
    Pratique de MySQL et PHP
</a>
```

À chaque lien dans un document HTML est associée une URL qui donne la localisation de la ressource. Les navigateurs permettent à l'utilisateur de suivre un

lien par simple clic de souris, et se chargent de récupérer le document correspondant grâce à l'URL. Ce mécanisme rend transparentes dans la plupart des cas, les adresses des documents web pour les utilisateurs.

Le Web peut être vu comme un immense, et très complexe, graphe de documents reliés par des liens hypertextes. Les liens présents dans un document peuvent donner accès non seulement à d'autres documents du même site, mais également à des documents gérés par d'autres sites, n'importe où sur le réseau.

Clients web

Le Web est donc un ensemble de serveurs connectés à l'Internet et proposant des ressources. L'utilisateur qui accède à ces ressources utilise en général un type particulier de programme client, le *navigateur*. Les deux principales tâches d'un navigateur consistent à :

1. dialoguer avec un serveur ;
2. afficher à l'écran les documents transmis par un serveur.

Les navigateurs offrent des fonctionnalités bien plus étendues que les deux tâches citées ci-dessus. Firefox dispose par exemple d'un mécanisme d'extension par *plugin* qui permet d'intégrer très facilement de nouveaux modules.

1.1.3 Programmation web

La programation web permet de dépasser les limites étroites des pages HTML statiques, dont le contenu est fixé à l'avance. Le principe consiste à produire les documents HTML par un programme associé au serveur web. Ce programme reçoit en outre des paramètres saisis par l'utilisateur qui conditionnent la page renvoyée par le serveur au client. Le contenu des pages est donc construit à la demande, « dynamiquement ».

La figure 1.2 illustre les composants de base d'une application web. Le navigateur (client) envoie une requête (souvent à partir d'un *formulaire* HTML). Cette requête consiste à déclencher une *action* (que nous désignons par « programme web » dans ce qui suit) sur un serveur référencé par son URL. L'exécution du programme web par le serveur web se déroule en trois phases :

1. *Constitution de la requête par le client* : le navigateur construit une URL contenant le nom du programme à exécuter, accompagné, le plus souvent, de paramètres ;
2. *Réception de la requête par le serveur* : le programme serveur récupère les informations transmises par le navigateur et déclenche l'exécution du programme en lui fournissant les paramètres reçus ;
3. *Transmission de la réponse* : le programme renvoie le résultat de son exécution au serveur sous la forme d'un document HTML, le serveur se contentant alors de faire suivre au client.

Nous décrivons brièvement ces trois étapes dans ce qui suit.

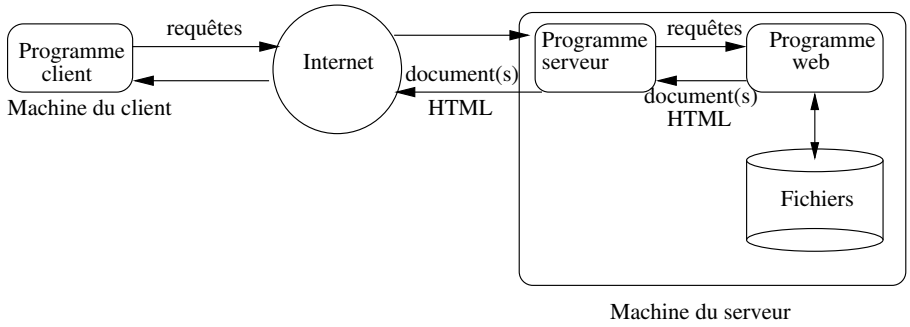


Figure 1.2 — Architecture basique d'une application web

Constitution d'une requête : les formulaires

Une requête transmise à un programme web est simplement une URL référençant le programme sur la machine serveur. On peut en théorie créer cette URL manuellement. On peut aussi la trouver intégrée à une ancre HTML. Mais le plus souvent, le programme déclenché sur le serveur doit recevoir des paramètres, et leur saisie est une tâche fastidieuse si elle ne se fait pas à l'aide d'un formulaire HTML.

Un formulaire est un conteneur HTML constitué d'un ensemble de balises définissant des champs de saisie. Les formulaires offrent la possibilité appréciable de créer très facilement une interface. En raison de leur importance, nous allons rappeler ici leurs principales caractéristiques en prenant l'exemple de la figure 1.3, qui montre un formulaire permettant la saisie de la description d'un film.

Différents types de champs sont utilisés :

- le titre et l'année sont des simples champs de saisie. L'utilisateur est libre d'entrer toute valeur alphanumérique de son choix ;
- le pays producteur est proposé sous la forme d'une liste de valeurs pré-définies. Le choix est de type exclusif : on ne peut cocher qu'une valeur à la fois ;
- le genre est lui aussi présenté sous la forme d'une liste de choix imposés, mais ici il est possible de sélectionner plusieurs choix simultanément ;
- l'internaute peut transmettre au serveur un fichier contenant l'affiche du film, grâce à un champ spécial qui offre la possibilité de choisir (bouton *Parcourir*) le fichier sur le disque local ;
- une liste présentée sous la forme d'un menu déroulant propose une liste des metteurs en scène ;
- on peut entrer le résumé du film dans une zone de saisie de texte ;
- enfin, les boutons « Valider » ou « Annuler » sont utilisés pour, au choix, transmettre les valeurs saisies au programme web, ou ré-initialiser le formulaire.

Cet exemple couvre pratiquement l'ensemble des types de champs disponibles. Nous décrivons dans ce qui suit les balises de création de formulaires.

Figure 1.3 — Présentation d'un formulaire avec Firefox

La balise <form>

C'est un conteneur délimité par <**form**> et </form> qui, outre les champs de saisie, peut contenir n'importe quel texte ou balise HTML. Les trois attributs suivants sont essentiels :

- `action` est la référence au programme exécuté par le serveur ;
- `method` indique le mode de transmission des paramètres au programme, avec essentiellement deux valeurs possibles, `get` ou `post` ;
- `enctype` indique le type d'encodage des données du formulaire, utilisé pour la transmission au serveur. Il y a deux valeurs possibles.

  1. *application/x-www-form-urlencoded.*
     Il s'agit de l'option par défaut, utilisée même quand on ne donne pas d'attribut `enctype`. Les champs du formulaire sont transmis sous la forme d'une liste de paires *nom=valeur*, séparées par des « & ».

  2. *multipart/form-data.*
     Cette option doit être utilisée pour les transmissions comprenant des fichiers. Le mode de transmission par défaut est en effet inefficace pour les fichiers à cause du codage assez volumineux utilisé pour les caractères non-alphanumériques. Quand on utilise *multipart/form-data*, les fichiers sont transmis séparément des champs classiques, dans une représentation plus compacte.

Voici le code HTML donnant le début du formulaire de la figure 1.3. Le service associé à ce formulaire est le programme `Film.php` qui se trouve au même endroit

que le formulaire. La méthode `post` indique un mode particulier de passage des paramètres.

```
<form action='Film.php' method='post' enctype='multipart/form-
    data'>
```

À l'intérieur d'un formulaire, on peut placer plusieurs types de champs de saisie, incluant des valeurs numériques ou alphanumériques simples saisies par l'utilisateur, des choix multiples ou exclusifs parmi un ensemble de valeurs pré-définies, du texte libre ou la transmission de fichiers.

La balise <input>

La balise <**input**> est la plus générale. Elle permet de définir tous les champs de formulaires, à l'exception des listes de sélection et des fenêtres de saisie de texte.

Chaque champ <**input**> a un attribut `name` qui permet, au moment du passage des paramètres au programme, de référencer les valeurs saisies sous la forme de couples *`nom=valeur`*. La plupart des champs ont également un attribut `value` qui permet de définir une valeur par défaut (voir exemple ci-dessous). Les valeurs de `name` ne sont pas visibles dans la fenêtre du navigateur : elles ne servent qu'à référencer les valeurs respectives de ces champs au moment du passage des paramètres au programme.

Le type d'un champ est défini par un attribut `type` qui peut prendre les valeurs suivantes :

`type='text'` Correspond à un champ de saisie permettant à l'utilisateur d'entrer une chaîne de caractères. La taille de l'espace de saisie est fixée par l'attribut `size`, et la longueur maximale par l'attribut `maxlength`. Voici le champ pour la saisie du titre du film.

```
Titre : <input type='text' size='20' name='titre'/>
```

Un paramètre `titre=Le+Saint` sera passé par le serveur au programme si l'utilisateur saisit le titre « Le Saint ».

`type='password'` Identique au précédent, mais le texte saisi au clavier n'apparaît pas en clair (une étoile '*' sera affichée par le navigateur en lieu et place de chaque caractère). Ce type de champ est principalement utile pour la saisie de mots de passe.

`type='hidden'` Un champ de ce type n'est pas visible à l'écran. Il est destiné à définir un paramètre dont la valeur est fixée, et à passer ce paramètre au programme en même temps que ceux saisis par l'utilisateur.

Par exemple le champ ci-dessous permet de passer systématiquement un paramètre `monNom` ayant la valeur `ExForm1`, pour indiquer au programme le nom du formulaire qui lui transmet les données saisies.

```
<input type='hidden' name='monNom' value='ExForm1'/>
```

Il est important de noter que « caché » ne veut pas dire « secret » ! Rien n'empêche un utilisateur de consulter la valeur d'un champ caché en regardant le code source du document HTML.

`type='checkbox'` Ce type crée les boutons associés à des valeurs, ce qui permet à l'utilisateur de cocher un ou plusieurs choix, sans avoir à saisir explicitement chaque valeur. En associant le *même* nom à un ensemble de champs `checkbox`, on indique au navigateur que ces champs doivent être groupés dans la fenêtre d'affichage.

L'exemple ci-dessous montre comment donner le choix (non exclusif) entre les genres des films.

```
Comédie   : <input type='checkbox' name='genre' value='C'/>
Drame     : <input type='checkbox' name='genre' value='D'/>
Histoire  : <input type='checkbox' name='genre' value='H'/>
Suspense  : <input type='checkbox' name='genre' value='S'/>
```

Contrairement aux champs de type `text` ou apparenté, les valeurs (champ `value`) ne sont pas visibles. On peut donc utiliser une codification ('C', 'D', ...) plus concise que les libellés descriptifs (Comédie, Drame, ...). Au moment où le formulaire sera validé, une information `genre=`*`valeur`* sera passée au programme pour chaque bouton sélectionné par l'utilisateur. Le programme est bien entendu supposé connaître la signification des codes qu'il reçoit.

`type='radio'` Comme précédemment, on donne le choix entre plusieurs valeurs, mais ce choix est maintenant exclusif. Par exemple on n'autorise qu'un seul pays producteur.

```
France      : <input type='radio' name='pays' value='FR'
    checked='1'/>
Etats-Unis  : <input type='radio' name='pays' value='US'/>
Allemagne   : <input type='radio' name='pays' value='DE'/>
Japon       : <input type='radio' name='pays' value='JP'/>
```

L'attribut `checked` permet de présélectionner un des choix. Il est particulièrement utile pour les champs `radio` mais peut aussi être utilisé avec les champs `checkbox`.

`type='submit'` Ce champ correspond en fait à un bouton qui valide la saisie et déclenche le programme sur le serveur. En principe, il n'y a qu'un seul bouton `submit`, mais on peut en utiliser plusieurs, chacun étant alors caractérisé par un attribut `name` auquel on associe une valeur spécifique.

```
<input type='submit' value='Valider'/>
```

La valeur de l'attribut `value` est ici le texte à afficher. Au lieu de présenter un bouton simple, on peut utiliser une image quelconque, avec le type `image`. Par exemple :

```
<input type='image' src='bouton.gif'/>
```

`type='reset'` Ce type est complémentaire de `submit`. Il indique au navigateur de ré-initialiser le formulaire.

`type='file'` On peut transmettre des fichiers par l'intermédiaire d'un formulaire. Le champ doit alors contenir le chemin d'accès au fichier sur l'ordinateur du client. Le navigateur associe au champ de type `file` un bouton permettant de sélectionner le fichier à transmettre au serveur pour qu'il le passe au programme. Les attributs sont identiques à ceux du type `text`.

Voici la définition du bouton permettant de transmettre un fichier contenant l'affiche du film.

```
<input type='file' size='20' name='affiche'/>
```

Il est possible d'indiquer la taille maximale du fichier à transférer en insérant un champ caché, de nom `max_file_size`, avant le champ `file`. L'attribut `value` indique alors sa taille.

```
<input type='hidden' name='max_file_size' value='100000'/>

max_file_size
```

La balise <select>

Le principe de ce type de champ est similaire à celui des champs `radio` ou `checkbox`. On affiche une liste d'options parmi lesquelles l'utilisateur peut faire un ou plusieurs choix. Le champ `select` est surtout utile quand le nombre de valeurs est élevé.

<**select**> est un conteneur dans lequel on doit énumérer, avec les balises <**option**>, tous les choix possibles. La balise <**option**> a elle-même un attribut `value` qui indique la valeur à envoyer au programme quand le choix correspondant est sélectionné par l'utilisateur. Voici par exemple un champ <**select**> proposant une liste de réalisateurs :

```
Metteur en scène :
<select name='realisateur' size='3'>
<option value='1'>Alfred Hitchcock</option>
<option value='2'>Maurice Pialat</option>
<option value='3' selected='1'>Quentin Tarantino</option>
<option value='4'>Akira Kurosawa</option>
<option value='5'>John Woo</option>
<option value='6'>Tim Burton</option>
</select>
```

L'attribut **size** indique le nombre de choix à visualiser simultanément. Par défaut, <**select**> propose un choix exclusif. L'attribut **multiple** donne la possibilité de sélectionner plusieurs choix.

Au niveau de la balise **option**, l'attribut **selected** permet de présélectionner un des choix (ou plusieurs si le champ <**select**> est de type **multiple**). Noter que si on sélectionne 'John Woo', la valeur 5 sera envoyée au programme pour le paramètre nommé `realisateur`. Le programme est supposé averti de la signification de ces codes.

La balise <textarea>

Enfin, la dernière balise des formulaires HTML, <**textarea**>, permet à l'utilisateur de saisir un texte libre sur plusieurs lignes. Ses principaux attributs, outre **name** qui permet de référencer le texte saisi, sont **cols** et **rows** qui indiquent respectivement le nombre de colonnes et de lignes de la fenêtre de saisie.

<**textarea**> est un conteneur : tout ce qui est placé entre les balises de début et de fin est proposé comme texte par défaut à l'utilisateur. Voici le code HTML du champ destiné à saisir le résumé du film :

```
<textarea name='resume' cols='30' rows='3'>Résumé du film</
    textarea>
```

L'exemple 1.2 donne le code HTML complet pour la création du formulaire de la figure 1.3. Une première remarque est que le code est long, relativement fastidieux à lire, et assez répétitif. En fait, lil est principalement constitué de balises HTML (ouvertures, fermetures, attributs), spécifique à l'application étant minoritaire. HTML est un langage « bavard », et une page HTML devient rapidement très volumineuse, confuse, et donc difficile à faire évoluer. Un des buts que nous nous fixerons avec PHP est de nous doter des moyens d'obtenir un code beaucoup plus concis.

Exemple 1.2 *exemples/ExForm1.html* : *Le formulaire complet*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Formulaire complet</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<form action='Film.php'
      method='post' enctype='multipart/form-data'>

  <input type='hidden' name='monNom' value='ExForm1'/>

  Titre : <input type='text' size='20' name='titre'/>
  Années : <input type='text' size='4' maxlength='4'
                  name='annee' value='2008'/>
  <p>
  Comédie   : <input type='checkbox' name='genre' value='C'/>
  Drame     : <input type='checkbox' name='genre' value='D'/>
  Histoire  : <input type='checkbox' name='genre' value='H'/>
  Suspense  : <input type='checkbox' name='genre' value='S'/>
  </p>
  <p>
   France     : <input type='radio' name='pays' value='FR'
       checked='1'/>
```

```
    Etats-Unis : <input type='radio' name='pays' value='US'/>
    Allemagne  : <input type='radio' name='pays' value='DE'/>
    Japon      : <input type='radio' name='pays' value='JP'/>
  </p><p>
  Affiche du film : <input type='file' size='20' name='affiche'/>
  </p>
  <p>
  Metteur en scène :
  <select name='realisateur' size='3'>
  <option value='1'>Alfred Hitchcock</option>
  <option value='2'>Maurice Pialat</option>
  <option value='3' selected='1'>Quentin Tarantino</option>
  <option value='4'>Akira Kurosawa</option>
  <option value='5'>John Woo</option>
  <option value='6'>Tim Burton</option>
  </select>
  <br/>
  <p>
   Résumé:
    <textarea name='resume' cols='30' rows='3'>Résumé du film
     </textarea>
  </p>
  <h1>Votre choix</h1>
  <input type='submit' value='Valider'/>
  <input type='reset' value='Annuler'/>
</form>

</body>
</html>
```

Deuxième remarque, qui vient à l'appui de la précédente : malgré tout le code employé, le résultat en terme de présentation graphique reste peu attrayant. Pour bien faire, il faudrait (au minimum) aligner les libellés et les champs de saisie, ce qui peut se faire avec un tableau à deux colonnes. Il est facile d'imaginer le surcroît de confusion introduit par l'ajout des balises <**table**>, <**tr**>, etc. Là encore l'utilisation de PHP permettra de produire du HTML de manière plus raisonnée et mieux organisée.

Enfin il n'est pas inutile de signaler que l'interface créée par un formulaire HTML est assez incomplète en termes d'ergonomie et de sécurité. Il faudrait pouvoir aider l'utilisateur dans sa saisie, et surtout contrôler que les valeurs entrées respectent certaines règles. Rien n'interdit, dans l'exemple donné ci-dessus, de ne pas entrer de titre pour le film, ou d'indiquer -768 pour l'année. Ce genre de contrôle peut être fait par le serveur *après* que l'utilisateur a validé sa saisie, mais ce mode de fonctionnement a l'inconvénient de multiplier les échanges sur le réseau. Le langage Javascript permet d'effectuer des contrôles au moment de la saisie; et un mécanisme nommé Ajax peut même être utilisé pour communiquer avec le serveur sans réafficher la page. Nous vous renvoyons à un livre consacré à ces techniques pour plus d'information.

Transmission de la requête du client au serveur

Tous les paramètres saisis dans un formulaire doivent maintenant être transmis au serveur web ou, plus précisément, au programme web sur ce serveur. Inversement, ce programme doit produire un document (généralement un document HTML) et le transmettre au navigateur *via* le serveur web.

Au moment où l'utilisateur déclenche la recherche, le navigateur concatène dans une chaîne de caractères une suite de descriptions de la forme *nomChamp=val* où *nomChamp* est le nom du champ dans le formulaire et *val* la valeur saisie. Les différents champs sont séparés par « & » et les caractères blancs sont remplacés par des « + ».

Par exemple, si on a saisi « Le Saint » dans `titre`, 1978 dans `annee` et coché le choix « Comédie », la chaîne est constituée par :

```
titre=Le+Saint&annee=1978&genre=C
```

Il existe alors deux manières de transmettre cette chaîne au serveur, selon que la méthode utilisée est `get` ou `post`.

1. **Méthode** `get` : la chaîne est placée à la fin de l'URL appelée, après un caractère '?'. On obtiendrait dans ce cas :
   ```
   http://localhost/Films.php?titre=Le+Saint&annee=1978&genre=C
   ```
2. **Méthode** `post` : la chaîne est transmise séparément de l'URL.

La méthode `post` est généralement préférée quand les paramètres à transmettre sont volumineux, car elle évite d'avoir à gérer des URL très longues. Elle présente cependant un inconvénient : quand on veut revenir, avec le bouton « Retour » du navigateur, sur une page à laquelle on a accédé avec `post`, le navigateur resoumet le formulaire. Bien sûr, il demande auparavant l'autorisation de l'utilisateur, mais ce dernier n'a pas en général de raison d'être conscient des inconvénients possibles d'une double (ou triple, ou quadruple) soumission.

Quand le serveur reçoit une requête, il lance le programme et lui transmet un ensemble de paramètres correspondant non seulement aux champs du formulaire, mais également à diverses informations relatives au client qui a effectué la requête. On peut par exemple savoir si l'on a affaire à Firefox ou à Safari. Ces transmissions de paramètres se font essentiellement par des variables d'environnement qui peuvent être récupérées par ce programme. Quand la méthode utilisée est `get`, une de ces variables (`QUERY_STRING`) contient la liste des paramètres issus du formulaire. Les variables les plus importantes sont décrites dans la table 1.1.

Le programme est en général écrit dans un langage spécialisé (comme PHP) qui s'intègre étroitement au programme serveur et facilite le mode de programmation particulier aux applications web. En particulier le langage offre une méthode simple pour récupérer les paramètres de la requête et les variables d'environnement. Il est libre de faire toutes les opérations nécessaires pour satisfaire la demande (dans la limite de ses droits d'accès bien sûr). Il peut notamment rechercher et transmettre des fichiers ou des images, effectuer des contrôles, des calculs, créer des rapports, etc. Il

peut aussi accéder à une base de données pour insérer ou rechercher des informations. C'est ce dernier type d'utilisation, dans sa variante PHP/MySQL, que nous étudions dans ce livre.

Tableau 1.1 — Variables d'environnement

Variable d'environnement	Description
REQUEST_METHOD	Méthode de transmission des paramètres (`get`, `post`, etc.).
QUERY_STRING	Une chaîne de caractères contenant tous les paramètres de l'appel en cas de méthode `get`. Cette chaîne doit être décodée (voir ci-dessus), ce qui constitue l'aspect le plus fastidieux du traitement.
CONTENT_LENGTH	Longueur de la chaîne transmise sur l'entrée standard, en cas de méthode `post`.
SERVER_NAME	Nom de la machine hébergeant le serveur web.
PATH_INFO	Informations sur les chemins d'accès menant par exemple vers des fichiers que l'on souhaite utiliser.
HTTP_USER_AGENT	Type et version du navigateur utilisé par le client.
REMOTE_ADDR	Adresse IP du client.
REMOTE_HOST	Nom de la machine du client.
REMOTE_USER	Nom de l'utilisateur, pour les sites protégés par une procédure d'identification (voir annexe A).
REMOTE_PASSWORD	Mot de passe de l'utilisateur, pour les sites sécurisés.

Transmission de la réponse du serveur au client

Le programme doit écrire le résultat sur sa sortie standard *stdout* qui, par un mécanisme de redirection, communique directement avec l'entrée standard *stdin* du serveur. Le serveur transmet alors ce résultat au client.

Ce résultat peut être n'importe quel document multimédia, ce qui représente beaucoup de formats, depuis le simple texte ASCII jusqu'à la vidéo. Dans le cas où la requête d'un client se limite à demander au serveur de lui fournir un fichier, le serveur se base sur l'extension de ce fichier pour déterminer son type.

Conformément au protocole HTTP, il faut alors transmettre ce type dans l'entête, avec la clause `Content-type: typeDocument`, pour que le navigateur sache comment décoder les informations qui lui proviennent par la suite. Pour un fichier HTML par exemple, l'extension est le plus souvent *.html*, et la valeur de *typeDocument* est `text/html`.

1.1.4 Sessions

Une caractéristique essentielle de la programmation web est son mode *déconnecté*. Le serveur ne mémorise pas les demandes successives effectuées par un client particulier, et ne peut donc pas tenir compte de l'historique des échanges pour améliorer la communication avec le client.

Un exemple familier qui illustre bien cette limitation est *l'identification* d'un visiteur sur un site. Cette identification se base sur un formulaire pour fournir un nom et un mot de passe, et le serveur, s'il valide ce code d'accès, peut alors donner le droit au visiteur de consulter des parties sensibles du site. Le problème est que lors de la prochaine connexion (qui peut avoir lieu dans les secondes qui suivent la première !) le serveur ne sera pas en mesure de reconnaître que le client s'est déjà connecté, et devra lui demander à nouveau nom et mot de passe.

L'absence de connexion permanente entre client et serveur web est un gros obstacle pour la gestion de *sessions* se déroulant dans un contexte riche, constitué d'un ensemble d'informations persistant tout au long des communications client/serveur. Par exemple, on voudrait stocker non seulement le nom et le mot de passe, mais aussi l'historique des accès au site afin de guider plus facilement un visiteur habitué. Plusieurs solutions, plus ou moins satisfaisantes, ont été essayées pour tenter de résoudre ce problème. La plus utilisée est de recourir aux *cookies*. Essentiellement, un *cookie* est une donnée, représentable sous la forme habituelle `nom=valeur`, que le navigateur conserve pour une période déterminée à la demande du serveur. Cette demande doit être effectuée dans un en-tête HTTP avec une instruction `Set-Cookie`. En voici un exemple :

```
Set-Cookie:
  MonCookie=200;
  expires=Mon,24-Dec-2010 12:00:00 GMT;
  path=/;
  domain=dauphine.fr
```

Cette instruction demande au navigateur de conserver jusqu'au 24 décembre 2010 un *cookie* nommé `MonCookie` ayant pour valeur 200. Les attributs optionnels `path` et `domain` restreignent la visibilité du *cookie* pour les programmes serveurs qui communiquent avec le navigateur. Par défaut, le *cookie* est transmis uniquement au serveur qui l'a défini, pour toutes les pages web gérées par lui. Ici on a élargi l'autorisation à tous les serveurs du domaine `dauphine.fr`.

Par la suite, les cookies stockés par un navigateur sont envoyés au serveur dans une variable d'environnement `HTTP_COOKIE`. Nous ne détaillons pas le format d'échange qui est relativement complexe à décrypter. PHP permet d'obtenir très facilement les *cookies*.

Il faut noter qu'un *cookie* ne disparaît pas quand le navigateur est stoppé puisqu'il est stocké dans un fichier. Il est toujours intéressant de consulter la liste des cookies (par exemple avec *Web Developer* pour voir qui a laissé traîner des informations chez vous. On peut considérer comme suspecte cette technique qui consiste à écrire des informations sur le disque dur d'un client à son insu, mais les *cookies* offrent le moyen le plus simple et puissant de créer un contexte persistant aux différents échanges client/serveur. Nous les utiliserons le moment venu pour gérer les sessions, par l'intermédiaire de fonctions PHP qui fournissent une interface simple et facile à utiliser.

1.2 PROGRAMMATION WEB AVEC MySQL ET PHP

Après cette introduction générale, nous en arrivons maintenant aux deux outils que nous allons associer pour développer des applications web avec simplicité et puissance.

1.2.1 MySQL

MySQL est un *Système de Gestion de Bases de Données* (SGBD) qui gère pour vous les fichiers constituant une base, prend en charge les fonctionnalités de protection et de sécurité et fournit un ensemble d'interfaces de programmation (dont une avec PHP) facilitant l'accès aux données.

La complexité de logiciels comme MySQL est due à la diversité des techniques mises en œuvre, à la multiplicité des composants intervenant dans leur architecture, et également aux différents types d'utilisateurs (administrateurs, programmeurs, non informaticiens, ...) confrontés, à différents niveaux, au système. Au cours de ce livre nous aborderons ces différents aspects, tous ne vous étant d'ailleurs pas utiles, en particulier si votre objectif n'est pas d'administrer une base MySQL. Pour l'instant, nous nous contenterons de décrire l'essentiel, à savoir son architecture et ses composants.

MySQL consiste en un ensemble de programmes chargés de gérer une ou plusieurs bases de données, et qui fonctionnent selon une architecture client/serveur (voir figure 1.4).

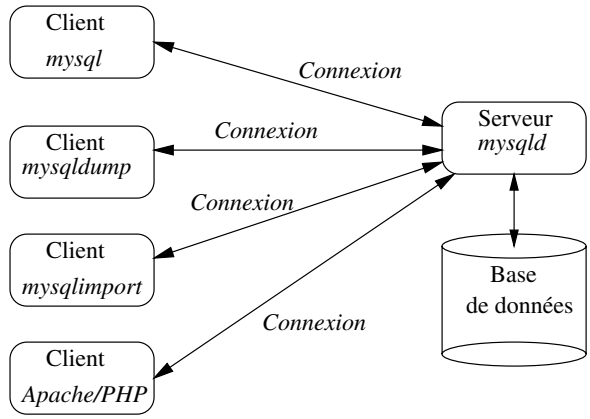


Figure 1.4 — Serveur et clients de MySQL.

Le serveur *mysqld*. Le processus *mysqld* est le serveur de MySQL. Lui seul peut accéder aux fichiers stockant les données pour lire et écrire des informations.

Utilitaires. MySQL fournit tout un ensemble de programmes, que nous appellerons *utilitaires* par la suite, chargés de dialoguer avec *mysqld*, par l'intermédiaire d'une *connexion*, pour accomplir un type de tâche particulier. Par exemple, *mysqldump* permet d'effectuer des sauvegardes, *mysqlimport* peut importer des fichiers ASCII dans une base, etc. Le client le plus simple est simplement nommé *mysql*, et permet d'envoyer directement des commandes au serveur.

La base de données est un ensemble de fichiers stockant les informations selon un format propre à MySQL et qui peut – doit – rester inconnu à l'utilisateur. Le serveur est le seul habilité à lire/écrire dans ces fichiers, en fonction de demandes effectuées par des clients MySQL. Plusieurs clients peuvent accéder simultanément à une même base. Le serveur se charge de coordonner ces accès.

Les clients de MySQL communiquent avec le serveur pour effectuer des recherches ou des mises à jour dans la base. Cette communication n'est pas limitée à des processus situés sur la même machine : il est possible de s'adresser au serveur MySQL par un réseau comme l'Internet. Dans une application PHP, le client est le serveur web (souvent Apache) qui n'est pas forcément situé sur la même machine que le processus *mysqld*.

Il est possible de créer soi-même son propre client MySQL en utilisant des outils de programmation qui se présentent sous la forme d'un ensemble de fonctions, habituellement désigné par l'acronyme API pour *Application Programming Interface*. MySQL fournit une API en langage C, à partir de laquelle plusieurs autres ont été créées, dont une API en PHP. Comme tous les autres clients de MySQL, un script PHP en association avec Apache doit établir une connexion avec le serveur pour pouvoir dialoguer avec lui et rechercher ou mettre à jour des données (figure 1.4).

Bases de données relationnelles

MySQL est un SGBD *relationnel*, comme beaucoup d'autres dont ORACLE, PostgreSQL, SQL Server, etc. Le point commun de tous ces systèmes est de proposer une représentation extrêmement simple de l'information sous forme de *table*. Voici une table relationnelle *Film*, donnant la description de quelques films.

titre	année	nom_realisateur	prénom_realisateur	annéeNaiss
Alien	1979	Scott	Ridley	1943
Vertigo	1958	Hitchcock	Alfred	1899
Psychose	1960	Hitchcock	Alfred	1899
Kagemusha	1980	Kurosawa	Akira	1910
Volte-face	1997	Woo	John	1946
Pulp Fiction	1995	Tarantino	Quentin	
Titanic	1997	Cameron	James	1954
Sacrifice	1986	Tarkovski	Andrei	1932

Il y a quelques différences essentielles entre cette représentation et le stockage dans un fichier. D'une part, les informations sont conformes à une description précise. Ici la table s'appelle *Film*, et elle comprend un ensemble d'attributs comme *titre*, *année*, etc. Une base de données est constituée d'une ou plusieurs tables, dont les descriptions sont connues et gérées par le serveur. Nous verrons qu'il est possible, *via* un langage simple, de spécifier le format d'une table, ainsi que le type des attributs et les contraintes qui s'appliquent aux données. Par exemple : *il ne doit pas exister deux films avec le même titre*. Tout ce qui concerne la description des données, et pas les données elles-mêmes, constitue le *schéma* de la base de données.

Les SGBD relationnels offrent non seulement une représentation simple et puissante, mais également un langage, SQL, pour interroger ou mettre à jour les données. SQL est incomparablement plus facile à utiliser qu'un langage de programmation classique comme le C. Voici par exemple comment on demande la liste des titres de film parus après 1980.

```
SELECT titre
FROM   Film
WHERE  annee > 1980
```

Cette approche très simple se contente d'indiquer ce que l'on veut obtenir, à charge pour le SGBD de déterminer *comment* on peut l'obtenir. SQL est un langage *déclaratif* qui permet d'interroger une base sans se soucier de la représentation interne des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires. À ce titre il s'adresse à une large communauté d'utilisateurs potentiels (pas seulement des informaticiens) et constitue un des atouts les plus spectaculaires (et le plus connu) des SGBD relationnels. On peut l'utiliser de manière interactive, mais également en association avec des interfaces graphiques, des outils de *reporting* ou, généralement, des langages de programmation.

Ce dernier aspect est important en pratique car SQL ne permet pas de faire de la programmation au sens courant du terme et doit donc être associé avec un langage comme PHP quand on souhaite effectuer des manipulations complexes.

1.2.2 PHP

Le langage PHP a été créé par Rasmus Lerdorf en 1994, pour ses besoins personnels. Comme dans beaucoup d'autres cas, la mise à disposition du langage sur l'Internet est à l'origine de son développement par d'autres utilisateurs qui y ont vu un outil propre à satisfaire leurs besoins. Après plusieurs évolutions importantes, PHP en est à sa version 5.2, celle que nous utilisons. La version 6 est annoncée à l'heure où ces lignes sont écrites. PHP – le plus souvent associé à MySQL – est à l'heure actuelle le plus répandu des langages de programmations pour sites web.

Qu'est-ce que PHP

PHP est un langage de programmation, très proche syntaxiquement du langage C, destiné à être intégré dans des pages HTML. Contrairement à d'autres langages, PHP est principalement dédié à la production de pages HTML générées dynamiquement. Voici un premier exemple.

Exemple 1.3 *exemples/ExPHP1.php* : *Premier exemple PHP*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>HTML avec PHP</title>
```

```
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>HTML + PHP</h1>

Nous sommes le <?php echo Date ("j/m/Y");  ?>

<p>

<?php
  echo "Je suis " .  $_SERVER['HTTP_USER_AGENT'] . "<br/>";
  echo "Je dialogue avec " . $_SERVER['SERVER_NAME'];
?>

</p>

</body>
</html>
```

Il s'agit d'un document contenant du code HTML classique, au sein duquel on a introduit des commandes encadrées par les balises `<?php` et `?>` (on appellera « scripts » les documents HTML/PHP à partir de maintenant). Tout ce qui se trouve entre ces commandes est envoyé à un interpréteur du langage PHP intégré à Apache. Cet interpréteur lit les instructions et les exécute.

> **REMARQUE –** Certaines configurations de PHP acceptent des balises PHP dites « courtes » (`<?` et `?>`) qui sont incompatibles avec XML et donc avec le langage XHTML que nous utilisons. Les scripts PHP écrits avec ces balises courtes ne sont pas portables : je vous déconseille fortement de les utiliser.

Ici on a deux occurrences de code PHP . La première fait partie de la ligne suivante :

```
Nous sommes le <?php echo Date ("j/m/Y"); ?>
```

Le début de la ligne est du texte traité par le serveur Apache comme du HTML. Ensuite, on trouve une instruction `echo Date ("j/m/Y");`. La fonction `echo()` est l'équivalent du `printf()` utilisé en langage C. Elle écrit sur la sortie standard, laquelle est directement transmise au navigateur par le serveur web. La fonction PHP `date()` récupère la date courante et la met en forme selon un format donné (ici, la chaîne `j/m/Y` qui correspond à jour, mois et année sur quatre chiffres).

La syntaxe de PHP est relativement simple, et la plus grande partie de la richesse du langage réside dans ses innombrables fonctions. Il existe des fonctions pour créer des images, pour générer du PDF, pour lire ou écrire dans des fichiers, et – ce qui nous intéresse particulièrement – pour accéder à des bases de données.

> **REMARQUE –** Le langage PHP est introduit progressivement à l'aide d'exemples. Si vous souhaitez avoir dès maintenant un aperçu complet du langage, reportez-vous au chapitre 11, page 419, qui en présente la syntaxe et peut se lire indépendamment.

Le script *ExPHP1.php* illustre un autre aspect essentiel du langage. Non seulement il s'intègre directement avec le langage HTML, mais toutes les variables d'environnement décrivant le contexte des communications entre le navigateur et le serveur web sont directement accessibles sous forme de variables PHP. Tous les noms de variables de PHP débutent par un « $ ». Voici la première ligne du script dans laquelle on insère une variable transmise par le serveur.

```
echo "Je suis ". $_SERVER['HTTP_USER_AGENT'] . "<br/>";
```

Le point « . » désigne l'opérateur de concaténation de chaîne en PHP. La commande `echo` envoie donc sur la sortie standard une chaîne obtenue en concaténant les trois éléments suivants :

- une sous-chaîne contenant le texte « `Je suis` »
- la chaîne correspondant au contenu de la variable `HTTP_USER_AGENT` ;
- la chaîne contenant la balise HTML <**br/**>.

Cette création de contenu par concaténation de texte simple, de variables PHP et de balises HTML est l'une des principales forces de PHP. Le point le plus important ici est l'exploitation de la variable `HTTP_USER_AGENT` qui représente le navigateur qui a demandé l'exécution du script. Cette variable est l'un des éléments du tableau PHP `$_SERVER` automatiquement créé par le serveur et transmis au script PHP. Ce tableau est de type « tableau associatif », chaque élément étant référencé par un nom. L'élément correspondant au nom du serveur est référencé par `SERVER_NAME` et se trouve donc accessible dans le tableau avec l'expression `$_SERVER['SERVER_NAME']`. C'est le cas de toutes les variables d'environnement (voir tableau 1.1, page 16).

Un script PHP a accès à plusieurs tableaux associatifs pour récupérer les variables d'environnement ou celles transmises via HTTP. La table 1.2 donne la liste de ces tableaux.

Tableau 1.2 — Tableaux prédéfinis de PHP

Tableau associatif	Contenu
`$_SERVER`	Contient les variables d'environnement énumérées dans la table 1.1, page 16 (comme `SERVER_NAME`, `CONTENT_TYPE`, etc), ainsi que des variables propres à l'environnement PHP comme `PHP_SELF`, le nom du script courant.
`$_ENV`	Contient les variables d'environnement système, que l'on peut également récupérer par la fonction `getenv()`.
`$_COOKIE`	Contient les *cookies* transmis au script.
`$_GET`	Contient les paramètres HTTP transmis en mode `get`.
`$_POST`	Contient les paramètres HTTP transmis en mode `post`.
`$_FILES`	Contient la liste des fichiers transmis au serveur par le navigateur.
`$_REQUEST`	Contient toutes les variables des quatre tableaux précédents.
`$_SESSION`	Contient les variables de session PHP.
`$GLOBALS`	Contient les variables globales du script.

En résumé, on dispose automatiquement, sous forme de variables PHP et sans avoir besoin d'effectuer un décryptage compliqué, de la totalité des informations échangées entre le client le serveur. Il faut noter que ces tableaux sont « globaux », c'est-à-dire accessibles dans toutes les parties du script, même au sein des fonctions ou des méthodes, sans avoir besoin de les passer en paramètres.

PHP est du côté serveur

Un script PHP est exécuté par un interpréteur situé *du côté serveur*. En cela, PHP est complètement différent d'un langage comme JavaScript, qui s'exécute sur le navigateur. En général l'interpréteur PHP est intégré à Apache sous forme de module, et le mode d'exécution est alors très simple. Quand un fichier avec une extension `.php`[1] est demandé au serveur web, ce dernier le charge en mémoire et y cherche tous les scripts PHP qu'il transmet à l'interpréteur. L'interpréteur exécute le script, ce qui a pour effet de produire du code HTML qui vient remplacer le script PHP dans le document finalement fourni au navigateur. Ce dernier reçoit donc du HTML « pur » et ne voit jamais la moindre instruction PHP.

À titre d'exemple, voici le code HTML produit par le fichier PHP précédent, tel que vous pouvez vous-mêmes le vérifier sur notre site. Le résultat correspond à une exécution sur la machine serveur *www.dauphine.fr* d'un script auquel on accède avec un navigateur Mozilla. Les parties HTML sont inchangées, le code PHP a été remplacé par le résultat des commandes `echo`.

```
<<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>HTML avec PHP</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>HTML + PHP</h1>

Nous sommes le 31/10/2008
<p>

Je suis Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; fr;
rv:1.9.0.3) G
ecko/2008092414 Firefox/3.0.3<br/>Je dialogue avec localhost
</p>

</body>
</html>
```

1. L'extension des scripts PHP (en général `.php`) est paramétrable dans le fichier *httpd.conf* de configuration d'Apache.

Accès à MySQL

Un des grands atouts de PHP est sa très riche collection d'interfaces (API) avec tout un ensemble de SGBD. En particulier, il est possible à partir d'un script PHP de se connecter à un serveur *mysqld* pour récupérer des données que l'on va ensuite afficher dans des documents HTML. D'une certaine manière, PHP permet de faire d'Apache un client MySQL, ce qui aboutit à l'architecture de la figure 1.5.

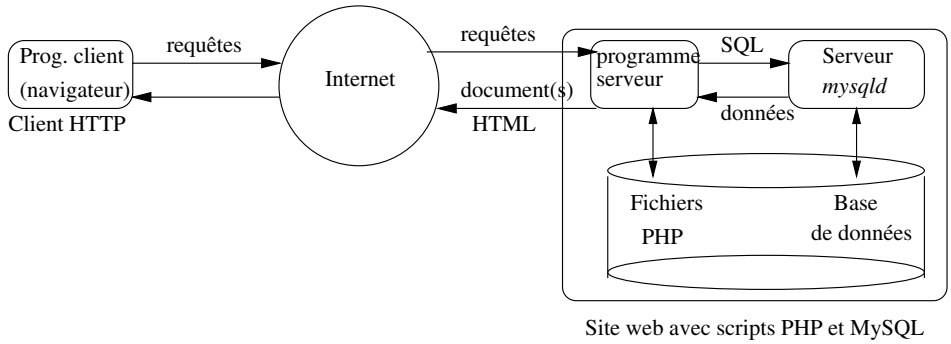


Figure 1.5 — Architecture d'un site web avec MySQL/PHP

Il s'agit d'une architecture à trois composantes, chacune réalisant l'une des trois tâches fondamentales d'une application.

1. le *navigateur* constitue *l'interface graphique* dont le rôle est de permettre à l'utilisateur de visualiser et d'interagir avec l'information ;
2. MySQL est *le serveur de données* ;
3. enfin, l'ensemble des fichiers PHP contenant le code d'extraction, traitement et mise en forme des données est le *serveur d'application*, associé à Apache qui se charge de transférer les documents produits sur l'Internet.

Rien n'empêche d'aller un tout petit peu plus loin et d'imaginer une architecture où les trois composantes sont franchement séparées et dialoguent par l'intermédiaire du réseau Internet. Ici, nous supposerons que le serveur *mysqld* et Apache sont sur la même machine, mais le passage à une solution réellement à « trois pôles » ne présente pas, ou peu, de différence du point de vue technique.

1.3 UNE PREMIÈRE BASE MySQL

Nous allons maintenant mettre ces principes en application en créant une première base MySQL contenant notre liste de films, et en accédant à cette base avec PHP. Pour l'instant nous présentons les différentes commandes d'une manière simple et intuitive avant d'y revenir plus en détail dans les prochains chapitres.

1.3.1 Création d'une table

La première base, très simple, est constituée d'une seule table *FilmSimple*, avec les quelques attributs déjà rencontrés précédemment. Pour créer des tables, on utilise une partie de SQL dite « Langage de Définition de Données » (LDD) dont la commande principale est `CREATE TABLE`.

```
CREATE TABLE FilmSimple
  (titre          VARCHAR (30),
   annee          INTEGER,
   nom_realisateur    VARCHAR (30),
   prenom_realisateur VARCHAR (30),
   annee_naissance INTEGER
  );
```

La syntaxe du `CREATE TABLE` se comprend aisément. On indique le nom de la table, qui sera utilisé par la suite pour accéder à son contenu, puis la liste des attributs avec leur type. Pour l'instant, nous nous en tenons à quelques types de base : `INTEGER`, que l'on peut abréger en `INT`, est un entier, et `VARCHAR` est une chaîne de caractères de longueur variable, pour laquelle on spécifie la longueur maximale.

> **REMARQUE –** On peut utiliser indifféremment les majuscules et les minuscules pour les mots-clés de SQL. De même, les sauts de ligne, les tabulations et les espaces successifs dans un ordre SQL équivalent à un seul espace pour l'interpréteur et peuvent donc être utilisés librement pour clarifier la commande.

Pour exécuter une commande SQL, il existe plusieurs possibilités, la plus générale étant d'utiliser le client *mysql* dont le rôle est principalement celui d'un interpréteur de commandes. Dans le cadre d'une application web, on dispose également d'une interface web d'administration de bases MySQL, *phpMyAdmin*. Cet outil fournit un environnement de travail *graphique*, plus convivial que l'interpréteur de commandes *mysql*.

Nous envisageons successivement les deux situations, *mysql* et *phpMyAdmin*, dans ce qui suit.

1.3.2 L'utilitaire *mysql*

Voici tout d'abord comment créer une base de données et un nouvel utilisateur avec l'utilitaire *mysql*.

```
% mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> CREATE DATABASE Films;
mysql>
mysql> GRANT ALL PRIVILEGES ON Films.* TO adminFilms@localhost
           IDENTIFIED BY 'mdpAdmin';
mysql> exit
```

Le prompt `mysql>` est celui de l'interpréteur de commandes de MySQL : ne pas entrer de commandes Unix à ce niveau, et réciproquement !

La commande `CREATE DATABASE` crée une base de données *Films*, autrement dit un espace dans lequel on va placer une ou plusieurs tables contenant les données de l'application. La commande `GRANT` définit un utilisateur `adminFilms` qui aura tous les droits (`ALL PRIVILEGES`) pour accéder à cette base et manipuler ses données. On peut alors se connecter à la base *Films* sous le compte `adminFilms` avec :

```
% mysql -u adminFilms -p Films
```

L'option `-p` indique que l'on veut entrer un mot de passe. *mysql* affiche alors un prompt

```
password:
```

Il est possible de donner le mot de passe directement après `-p` dans la ligne de commande mais ce n'est pas une très bonne habitude à prendre que d'afficher en clair des mots de passe.

La meilleure méthode consiste à stocker dans un fichier de configuration le compte d'accès à MySQL et le mot de passe. Pour éviter à l'utilisateur `adminFilms` d'entrer systématiquement ces informations, on peut ainsi créer un fichier *.my.cnf* dans le répertoire `$HOME` (sous Unix ou Mac OS, dans une fenêtre *terminal*) ou C: (sous Windows), et y placer les informations suivantes :

```
[client]
user= adminFilms
password = mdpAdmin
```

Tous les programmes clients de MySQL lisent ce fichier et utiliseront ce compte d'accès pour se connecter au programme serveur *mysqld*. La connexion à la base *Films* devient alors simplement :

```
% mysql Films
```

Bien entendu, il faut s'assurer que le fichier *.my.cnf* n'est pas lisible par les autres utilisateurs. Nous renvoyons à l'annexe A pour plus de détails sur l'utilisation des fichiers de configuration.

Création de la table

On suppose maintenant que l'utilisateur dispose d'un fichier de configuration. Voici la séquence complète de commandes pour créer la table *FilmSimple*.

```
% mysql

Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> USE Films;
```

```
Database changed
mysql> CREATE TABLE FilmSimple
    ->   (titre      VARCHAR (30),
    ->    annee      INTEGER,
    ->    nom_realisateur     VARCHAR (30),
    ->    prenom_realisateur  VARCHAR (30),
    ->    annee_naissance INTEGER
    ->   );
Query OK, 0 rows affected (0.01 sec)
mysql> exit
```

La commande `USE Films` indique que l'on s'apprête à travailler sur la base *Films* (rappelons qu'un serveur peut gérer plusieurs bases de données). On peut, de manière équivalente, donner le nom de la base sur la ligne de commande de *mysql*. La commande `CREATE TABLE`, entrée ligne à ligne, provoque l'affichage de `->` tant qu'un point-virgule indiquant la fin de la commande n'est pas saisie au clavier.

> **REMARQUE –** Peut-on utiliser des accents dans les noms de table et d'attributs ? La réponse est *oui,* du moins si MySQL a été installé en précisant que le jeu de caractères à utiliser est du type *Latin*. Il y a quand même un petit problème à prendre en compte : ces noms sont utilisés pour interroger la base de données, et tous les claviers ne disposent pas des caractères accentués. J'ai pris le parti de ne pas utiliser d'accent pour tout le code informatique. En revanche, les informations stockées dans la base pourront elles contenir des accents. À vous de juger du choix qui convient à votre situation.

La table *FilmSimple* est maintenant créée. Vous pouvez consulter son schéma avec la commande `DESCRIBE` (`DESC` en abrégé) et obtenir l'affichage ci-dessous. Seules les informations `Field` et `Type` nous intéressent pour l'instant (pour des raisons obscures, MySQL affiche `int(11)` au lieu de `INTEGER` dans la colonne `Type`...).

```
mysql> DESC FilmSimple;
+--------------------+-------------+------+-----+---------+-------+
| Field              | Type        | Null | Key | Default | Extra |
+--------------------+-------------+------+-----+---------+-------+
| titre              | varchar(30) | YES  |     | NULL    |       |
| annee              | int(11)     | YES  |     | NULL    |       |
| nom_realisateur    | varchar(30) | YES  |     | NULL    |       |
| prenom_realisateur | varchar(30) | YES  |     | NULL    |       |
| annee_naissance    | int(11)     | YES  |     | NULL    |       |
+--------------------+-------------+------+-----+---------+-------+
```

Pour détruire une table, on dispose de la commande `DROP TABLE`.

```
mysql> DROP TABLE FilmSimple;
Query OK, 0 rows affected (0.01 sec)
```

Il est assez fastidieux d'entrer au clavier toutes les commandes de *mysql*, d'autant que toute erreur de frappe implique de recommencer la saisie au début. Une meilleure

solution est de créer un fichier contenant les commandes et de l'exécuter. Voici le fichier *FilmSimple.sql* (nous parlerons de *script SQL* à partir de maintenant).

Exemple 1.4 *exemples/FilmSimple.sql* : *Fichier de création de la table FilmSimple*

```
/* Création d'une table 'FilmSimple'  */

CREATE TABLE FilmSimple
  (titre          VARCHAR (30),
   annee          INTEGER,
   nom_realisateur    VARCHAR (30),
   prenom_realisateur  VARCHAR (30),
   annee_naissance INTEGER
  );
```

Un script SQL peut contenir tout un ensemble de commandes, chacune devant se terminer par un ';'. Toutes les lignes commençant par « # », ou tous les textes encadrés par `/*`, `*/`, sont des commentaires.

On indique à *mysql* qu'il doit prendre ses commandes dans ce fichier au lieu de l'entrée standard de la manière suivante :

```
% mysql -u adminFilms -p < FilmSimple.sql
```

ou simplement, si on utilise un fichier de configuration avec nom et mot de passe :

```
% mysql < FilmSimple.sql
```

Le caractère « < » permet une redirection de l'entrée standard (par défaut la console utilisateur) vers *FilmSimple.sql*. Dernière solution, quand on est déjà sous l'interpréteur de MySQL, on peut exécuter les commandes contenues dans un fichier avec la commande `source` :

```
mysql> source FilmSimple.sql
```

Si l'on utilise un utilitaire comme PhpMyAdmin, le plus simple est de copier-coller la commande depuis le fichier vers la fenêtre adéquate de PhpMyAdmin (voir page 34).

Insertion de données

Nous avons maintenant une table *FilmSimple* dans laquelle nous pouvons insérer des données avec la commande SQL `INSERT`. Voici sa syntaxe :

```
INSERT INTO FilmSimple (titre, annee, prenom_realisateur,
    nom_realisateur)
VALUES ('Pulp Fiction', 1995,  'Quentin', 'Tarantino');
```

On indique la table dans laquelle on veut insérer une ligne, puis la liste des attributs auxquels ont va affecter une valeur. Les attributs qui n'apparaissent pas, comme,

pour cet exemple l'année de naissance du metteur en scène `annee_naissance`, auront une valeur dite `NULL` sur laquelle nous reviendrons plus tard.

La dernière partie de la commande `INSERT` est la liste des valeurs, précédée du mot-clé `VALUES`. Il doit y avoir autant de valeurs que d'attributs, et les chaînes de caractères doivent être encadrées par des apostrophes simples (’), ce qui permet d'y introduire des blancs.

> **REMARQUE –** MySQL accepte les apostrophes doubles ("), mais ce n'est pas conforme à la norme SQL ANSI. Il est préférable de prendre l'habitude d'utiliser systématiquement les apostrophes simples. Il n'est pas nécessaire d'utiliser des apostrophes pour les noms d'attributs, sauf s'ils correspondent à des mots-clés SQL, auquel cas on utilise l'apostrophe inversée (`).

Voici l'exécution de cette commande `INSERT` avec l'utilitaire *mysql*.

```
% mysql Films

Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> INSERT INTO FilmSimple
    -> (titre, annee, nom_realisateur, prenom_realisateur)
    ->  VALUES ('Pulp Fiction', 1995,  'Quentin', 'Tarantino');
Query OK, 1 row affected (0.16 sec)
```

La commande `INSERT` est en fait rarement utilisée directement, car sa syntaxe est assez lourde et surtout il n'y a pas de contrôle sur les valeurs des attributs. Le plus souvent, l'insertion de lignes dans une table se fait avec l'une des deux méthodes suivantes.

Saisie dans une interface graphique. Ce type d'interface offre une aide à la saisie, permet des contrôles, et facilite la tâche de l'utilisateur. Nous verrons comment créer de telles interfaces sous forme de formulaires HTML.

Chargement « en masse » à partir d'un fichier. Dans ce cas un programme lit les informations dans le fichier contenant les données, et effectue répétitivement des ordres `INSERT` pour chaque ligne trouvée.

MySQL fournit une commande accessible par l'interpréteur de commande, `LOAD DATA`, qui évite dans beaucoup de cas d'avoir à écrire un programme spécifique pour charger des fichiers dans une base. Cette commande est capable de lire de nombreux formats différents. Nous prenons comme exemple le fichier *films.txt*, fourni avec les exemples, dont voici le contenu :

```
Alien 1979 Scott Ridley 1943
Vertigo 1958 Hitchcock Alfred 1899
Psychose 1960 Hitchcock Alfred 1899
Kagemusha 1980 Kurosawa Akira 1910
```

```
Volte-face 1997 Woo John 1946
Titanic 1997 Cameron James 1954
Sacrifice 1986 Tarkovski Andrei 1932
```

Voici comment on utilise la commande `LOAD DATA` pour insérer en une seule fois le contenu de *films.txt* dans la table *FilmSimple*.

Exemple 1.5 *exemples/LoadData.sql* : *Commande de chargement d'un fichier dans la base*

```
# Chargement du fichier films.txt dans la table FilmSimple

LOAD DATA LOCAL INFILE  'films.txt'
INTO TABLE  FilmSimple
FIELDS TERMINATED BY ' ';
```

Voici quelques explications sur les options utilisées :

- l'option `LOCAL` indique au serveur que le fichier se trouve sur la machine du client *mysql*. Par défaut, le serveur cherche le fichier sur sa propre machine, dans le répertoire contenant la base de données ;
- on donne ici simplement le nom '*films.txt*', *ce qui suppose que le client mysql a été lancé dans le répertoire où se trouve ce fichier*. Si ce n'est pas le cas, il faut indiquer le chemin d'accès complet.
- enfin, il existe de nombreuses options pour indiquer le format du fichier. Ici on indique qu'une ligne dans le fichier correspond à une ligne dans la table, et que les valeurs des attributs dans le fichier sont séparées par des blancs.

Il s'agit d'un format simple mais notablement insuffisant. Il n'est pas possible d'avoir des blancs dans le titre de films, ou d'ignorer la valeur d'un attribut. On ne saurait pas charger la description du film *Pulp Fiction* avec ce format. Heureusement `LOAD DATA` sait traiter des formats de fichier beaucoup plus complexes. Une description de cette commande est donnée dans l'annexe B, page 470.

L'exécution sous *mysql* donne le résultat suivant :

```
% mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> LOAD DATA LOCAL INFILE  'films.txt'
    -> INTO TABLE  FilmSimple
    -> FIELDS TERMINATED BY ' ';
Query OK, 7 rows affected (0.00 sec)
Records: 7  Deleted: 0  Skipped: 0  Warnings: 0
```

Interrogation et modification

Le langage SQL propose les quatre opérations essentielles de manipulation de données : *insertion*, *destruction*, *mise à jour* et *recherche*. Ces opérations sont communément désignées par le terme de *requêtes*. L'ensemble des commandes permettant d'effectuer

ces opérations est le *Langage de Manipulation de Données*, ou LMD, par opposition au *Langage de Définition de Données* ou LDD.

Nous avons déjà vu la commande `INSERT` qui effectue des insertions. Nous introduisons maintenant les trois autres opérations en commençant par la recherche qui est de loin la plus complexe. Les exemples qui suivent s'appuient sur la table *FilmSimple* contenant les quelques lignes insérées précédemment.

```
SELECT  titre , annee
FROM    FilmSimple
WHERE   annee > 1980
```

Ce premier exemple nous montre la structure de base d'une recherche avec SQL, avec les trois clauses `SELECT`, `FROM` et `WHERE`.

- `FROM` indique la table dans laquelle on trouve les attributs utiles à la requête. Un attribut peut être « utile » de deux manières (non exclusives) : (1) on souhaite afficher son contenu (clause `SELECT`), (2) il a une valeur particulière (une constante ou la valeur d'un autre attribut) que l'on indique dans la clause `WHERE`.
- `SELECT` indique la liste des attributs constituant le résultat.
- `WHERE` indique les conditions que doivent satisfaire les lignes de la base pour faire partie du résultat.

> **REMARQUE –** sous Unix, une table comme *FilmSimple* est stockée par MySQL dans un fichier de nom *FilmSimple*. Comme Unix distingue les majuscules et les minuscules pour les noms de fichiers, il faut absolument respecter la casse dans le nom des tables, sous peine d'obtenir le message `Table does not exist`.

Cette requête peut être directement effectuée sous *mysql*, ce qui donne le résultat suivant.

```
mysql> SELECT  titre, annee
    -> FROM    FilmSimple
    -> WHERE    annee > 1980;
+------------+-------+
| titre      | annee |
+------------+-------+
| Volte-face |  1997 |
| Titanic    |  1997 |
| Sacrifice  |  1986 |
+------------+-------+
3 rows in set (0.00 sec)
```

N'oubliez pas le point-virgule pour finir une commande. La requête SQL la plus simple est celle qui affiche toute la table, sans faire de sélection (donc sans clause `WHERE`) et en gardant tous les attributs. Dans un tel cas on peut simplement utiliser le caractère « * » pour désigner la liste de tous les attributs.

```
mysql> SELECT * FROM FilmSimple;
+------------+-------+-----------------+--------------------+-----------------+
| titre      | annee | nom_realisateur | prenom_realisateur | annee_naissance |
+------------+-------+-----------------+--------------------+-----------------+
| Alien      |  1979 | Scott           | Ridley             |            1943 |
| Vertigo    |  1958 | Hitchcock       | Alfred             |            1899 |
| Psychose   |  1960 | Hitchcock       | Alfred             |            1899 |
| Kagemusha  |  1980 | Kurosawa        | Akira              |            1910 |
| Volte-face |  1997 | Woo             | John               |            1946 |
| Titanic    |  1997 | Cameron         | James              |            1954 |
| Sacrifice  |  1986 | Tarkovski       | Andrei             |            1932 |
+------------+-------+-----------------+--------------------+-----------------+
7 rows in set (0.00 sec)
```

Les requêtes les plus complexes que l'on puisse faire à ce stade sont celles qui sélectionnent des films selon des critères comme « Les films dont le titre est *Vertigo*, ou dont le prénom du metteur en scène est *John*, ou qui sont parus dans les années 90 ». La clause `WHERE` permet la combinaison de ces critères avec les connecteurs `AND`, `OR` et l'utilisation éventuelle des parenthèses pour lever les ambiguïtés.

```
mysql> SELECT titre, annee
    -> FROM  FilmSimple
    -> WHERE titre = 'Vertigo'
    -> OR    prenom_realisateur = 'Alfred'
    -> OR     (annee >= 1990 AND annee < 2000);
+------------+-------+
| titre      | annee |
+------------+-------+
| Vertigo    |  1958 |
| Psychose   |  1960 |
| Volte-face |  1997 |
| Titanic    |  1997 |
+------------+-------+
4 rows in set (0.00 sec)
```

Tant qu'il n'y a qu'une table à interroger, l'utilisation de SQL s'avère extrêmement simple. Le serveur fait tout le travail pour nous : accéder au fichier, lire les lignes, retenir celles qui satisfont les critères, satisfaire simultanément (ou presque) les demandes de plusieurs utilisateurs, etc. Dès qu'on interroge une base avec plusieurs tables, ce qui est la situation normale, les requêtes SQL deviennent un peu plus complexes.

> **REMARQUE –** Comme pour PHP, nous introduisons SQL au fur et à mesure. Les requêtes sur plusieurs tables (jointures) sont présentées dans le chapitre 7, page 289. Les requêtes d'agrégation sont présentées dans ce même chapitre, page 307. Enfin, le chapitre 10 propose un récapitulatif complet sur le langage.

Les commandes de mise à jour et de destruction sont des variantes du `SELECT`. On utilise la même clause `WHERE`, en remplaçant dans un cas le `SELECT` par `UPDATE`, et dans l'autre par `DELETE`. Voici deux exemples.

- *Détruire tous les films antérieurs à 1960.*
  Le critère de sélection des films à détruire est exprimé par une clause `WHERE`.
  ```
  DELETE FROM FilmSimple WHERE annee <= 1960
  ```
  Les données détruites sont *vraiment* perdues (sauf si vous utilisez le mode transactionnel de MySQL, optionnel). Ceux qui auraient l'habitude d'un système gérant les transactions doivent garder en mémoire qu'il n'y a pas de possibilité de retour en arrière avec `rollback` dans le fonctionnement par défaut de MySQL.
- *Changer le nom de 'John Woo' en 'Jusen Wu'.*
  La commande est légèrement plus complexe. On indique par une suite de `SET` *`attribut=valeur`* l'affectation de nouvelles valeurs à certains attributs des lignes modifiées.

  ```
  UPDATE FilmSimple SET nom_realisateur='Wu',
      prenom_realisateur='Yusen'
  WHERE nom_realisateur = 'Woo'
  ```

  Même remarque que précédemment : sauf en mode transactionnel, toutes les lignes sont modifiées sans possibilité d'annulation. Une manière de s'assurer que la partie de la table affectée par un ordre `DELETE` ou `UPDATE` est bien celle que l'on vise est d'effectuer au préalable la requête avec `SELECT` et la même clause `WHERE`.

Voici l'exécution sous *mysql*.

```
mysql> DELETE FROM FilmSimple WHERE annee <= 1960;
Query OK, 2 rows affected (0.01 sec)
mysql>
mysql>  UPDATE FilmSimple SET nom_realisateur='Wu',
                              prenom_realisateur='Yusen'
    ->  WHERE nom_realisateur = 'Woo';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Quelques commandes utiles

Enfin, *mysql* fournit tout un ensemble de commandes pour inspecter les tables, donner la liste des tables d'une base de données, etc. Voici une sélection des commandes les plus utiles. L'annexe B donne une liste exhaustive de toutes les fonctionnalités de MySQL.

- `SELECT DATABASE();` C'est une pseudo-requête SQL (sans `FROM`) qui affiche le nom de la base courante.
- `SELECT USER();` Idem, cette pseudo-requête affiche le nom de l'utilisateur courant.
- `SHOW DATABASES;` Affiche la liste des bases de données.
- `SHOW TABLES;` Affiche la liste des tables de la base courante.
- `SHOW COLUMNS FROM` *`NomTable`*`;` Affiche la description de la table *`NomTable`*.

1.3.3 L'interface PhpMyAdmin

PhpMyAdmin est un outil entièrement écrit en PHP qui fournit une interface simple et très complète pour administrer une base MySQL. La plupart des commandes de l'utilitaire *mysql* peuvent s'effectuer par l'intermédiaire de *phpMyAdmin*, les opérations possibles dépendant bien sûr des droits de l'utilisateur qui se connecte à la base. Voici une liste des principales possibilités :

1. Créer et détruire des bases de données (sous le compte `root` de MySQL).
2. Créer, détruire, modifier la description des tables.
3. Consulter le contenu des tables, modifier certaines lignes ou les détruire, etc.
4. Exécuter des requêtes SQL interactivement.
5. Charger des fichiers dans des tables et, réciproquement, récupérer le contenu de tables dans des fichiers ASCII.
6. Administrer MySQL.

Beaucoup de fournisseurs d'accès utilisent ce produit pour permettre la création, modification ou mise à jour d'une base de données personnelle à distance, à l'aide d'un simple navigateur. L'annexe A décrit l'installation de phpMyAdmin. Même s'il ne dispense pas complètement de l'utilisation de l'utilitaire *mysql*, il permet de faire beaucoup d'opérations simples de manière conviviale.

La figure 1.6 montre une copie d'écran de la page d'accueil de phpMyAdmin, après connexion d'un utilisateur. L'écran est divisé en deux parties. Sur la gauche un menu déroulant propose la liste des bases de données accessibles à l'utilisateur (si vous accédez au système d'un fournisseur d'accès, vous ne verrez certainement que votre base personnelle). Cette partie gauche reste affichée en permanence. La partie droite présente l'ensemble des opérations disponibles en fonction du contexte.

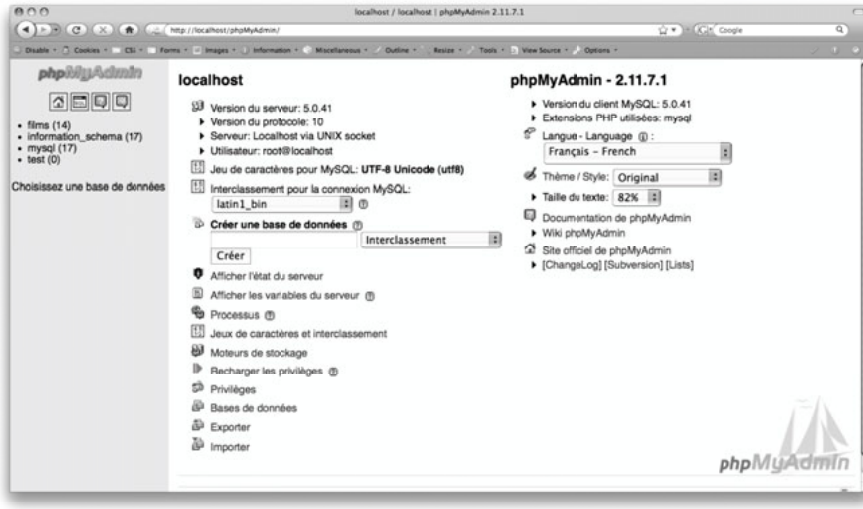


Figure 1.6 — Page d'accueil de phpMyAdmin

Initialement, si le compte de connexion utilisé est `root`, phpMyAdmin propose de consulter la situation du serveur et des clients MySQL, et des options de configuration de phpMyAdmin lui-même (notamment la langue).

En sélectionnant une des bases, on obtient sa structure (à savoir la liste des tables), et toute une liste d'actions à effectuer sur cette base. La figure 1.7 montre cette seconde page (noter qu'il s'agit d'un formulaire HTML). Voici quelques indications sur les fonctionnalités proposées :

Structure. Pour chaque table affichée, on peut effectuer les opérations suivantes.

1. *Afficher* donne le contenu de la table.
2. *Sélectionner* propose un petit formulaire permettant de sélectionner une partie de la table.
3. *Insérer* présente un autre formulaire, créé dynamiquement par phpMyAdmin, cette fois pour insérer des données dans la table.
4. *Propriétés* donne la description de la table et de ses index. Cette option donne accès à une autre page, assez complète, qui permet de modifier la table en ajoutant ou en supprimant des attributs.
5. *Supprimer* détruit la table (phpMyAdmin demande confirmation).
6. *Vide* détruit toutes les lignes.

SQL. La fenêtre placée en dessous de la liste des tables permet d'entrer des commandes SQL directement.

Pour créer la table *FilmSimple*, on peut copier/coller directement la commande `CREATE TABLE` dans cette fenêtre et l'exécuter. De même, on peut effectuer des `INSERT`, des `SELECT`, et toutes les commandes vues dans la section précédente.

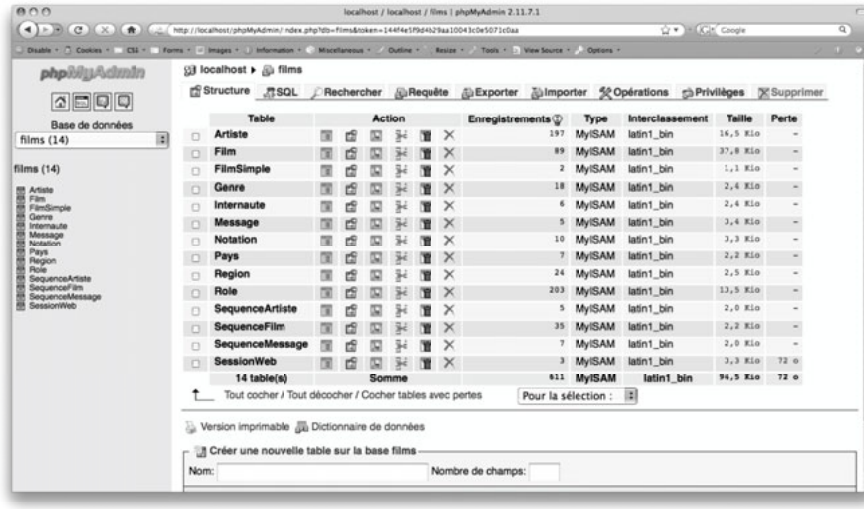


Figure 1.7 — Actions sur une base avec phpMyAdmin

Cette fenêtre est, dans phpMyAdmin, la fonctionnalité la plus proche de l'utilitaire *mysql*.

Exporter. Cette partie permet de créer un fichier contenant toutes les commandes de création de la base, ainsi que, optionnellement, les ordres d'insertion des données sous forme de commandes `INSERT`. En d'autres termes vous pouvez faire une sauvegarde complète, sous forme d'un fichier ASCII. En choisissant l'option *transmettre*, le fichier est transmis au navigateur.

Rechercher. Permet d'effectuer une recherche par mot-clé.

Requête. Donne accès à un formulaire aidant à la construction de requêtes SQL complexes, sans connaître SQL.

Supprimer. Supprime la base, avec toutes ses tables (après confirmation).

Enfin, le bas de cette page principale propose un formulaire pour créer une nouvelle table. Avant le bouton « Exécuter », il faut entrer le nom de la table et le nombre d'attributs.

L'utilisation de phpMyAdmin est simple et s'apprend en pratiquant. Bien que cet outil, en offrant une interface de saisie, économise beaucoup de frappe au clavier, il s'avère quand même nécessaire à l'usage de connaître les commandes SQL, ne serait-ce que pour comprendre les actions effectuées et les différentes options possibles. Dans tout ce qui suit, nous continuerons à présenter les commandes du langage SQL avec l'outil *mysql*, sachant qu'il suffit d'exécuter ces commandes dans la fenêtre `SQL` de phpMyAdmin pour obtenir le même résultat.

1.4 ACCÈS À MySQL AVEC PHP

Maintenant que nous disposons d'une base MySQL, nous pouvons aborder les outils d'accès à cette base à partir de scripts PHP. Nous étudions successivement dans cette section les aspects suivants :

L'interface fonctionnelle MySQL/PHP. Il s'agit d'un ensemble de fonctions qui, pour l'essentiel, permettent de se connecter à MySQL, d'exécuter des requêtes SQL et de récupérer le résultat que l'on peut ensuite afficher dans une page HTML.

Interrogation à partir de formulaires HTML. Nous montrons comment associer un formulaire et un programme interrogeant la base de données ;

Insertions et mises à jour. Toujours à partir de formulaires HTML, on peut créer des scripts PHP qui insèrent de nouvelles informations ou modifient celles qui existent déjà.

1.4.1 L'interface MySQL/PHP

PHP communique avec MySQL par l'intermédiaire d'un ensemble de fonctions qui permettent de récupérer, modifier, ou créer à peu près toutes les informations relatives à une base de données. Parmi ces informations, il faut compter bien entendu le contenu des tables, mais également leur description (le *schéma* de la base). L'utilitaire phpMyAdmin utilise par exemple les fonctions permettant d'obtenir le schéma pour présenter une interface d'administration, engendrer à la volée des formulaires de saisie, etc.

Le tableau 1.3 donne la liste des principales fonctions de l'API. Nous renvoyons à l'annexe C pour une liste exhaustive des fonctions MySQL/PHP.

Tableau 1.3 — Principales fonctions de l'API MySQL/PHP

Fonction	Description
`mysql_connect()`	Pour établir une connexion avec MySQL, pour un compte utilisateur et un serveur donnés. Renvoie une valeur utilisée ensuite pour dialoguer avec le serveur.
`mysql_pconnect()`	Idem, mais avec une connexion *persistante* (voir annexe C). Cette deuxième version est plus performante quand l'interpréteur PHP est inclus dans Apache.
`mysql_select_db()`	Permet de se placer dans le contexte d'une base de données. C'est l'équivalent de la commande `USE base` sous *mysql*.
`mysql_query()`	Pour exécuter une requête SQL ou n'importe quelle commande MySQL. Renvoie une variable représentant le résultat de la requête.
`mysql_fetch_object()`	Récupère une des lignes du résultat et positionne le curseur sur la ligne suivante. La ligne est représentée sous forme d'un *objet* (un groupe de valeurs).
`mysql_fetch_row()`	Récupère une des lignes du résultat, et positionne le curseur sur la ligne suivante. La ligne est représentée sous forme d'un *tableau* (une liste de valeurs).
`mysql_error()`	Renvoie le message de la dernière erreur rencontrée.

Voici maintenant ces fonctions en action. Le script suivant effectue une recherche de toutes les lignes de la table *FilmSimple* et affiche la liste des films dans une page HTML.

Exemple 1.6 *exemples/ExMyPHP1.php* : *Accès à MySQL avec PHP*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Connexion à MySQL</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
```

```
<body>

<h1>Interrogation de la table FilmSimple</h1>

<?php
require ("Connect.php");

$connexion = mysql_pconnect (SERVEUR, NOM, PASSE);

if (!$connexion) {
  echo "Désolé, connexion à " . SERVEUR . " impossible\n";
  exit;
}

if (!mysql_select_db (BASE, $connexion)) {
  echo "Désolé, accès à la base " . BASE . " impossible\n";
  exit;
}

$resultat = mysql_query ("SELECT * FROM FilmSimple", $connexion);

if ($resultat) {
  while ($film = mysql_fetch_object ($resultat)) {
    echo "$film->titre, paru en $film->annee, réalisé "
      . "par  $film->prenom_realisateur $film->nom_realisateur.<
          br/>\n";
  }
}
else {
  echo "<b>Erreur dans l'exécution de la requête.</b><br/>";
  echo "<b>Message de MySQL :</b> " .  mysql_error($connexion);
}
?>
</body>
</html>
```

Nous allons commenter soigneusement ce code qui est représentatif d'une bonne partie des techniques nécessaires pour accéder à une base MySQL et en extraire des informations. Le script proprement dit se réduit à la partie comprise entre les balises `<?php` et `?>`.

Inclusion de fichiers – Constantes

La première instruction est `require ("Connect.php");`

La commande `require` permet d'inclure le contenu d'un fichier dans le script. Certaines informations sont communes à beaucoup de scripts, et les répéter systématiquement est à la fois une perte de temps et une grosse source d'ennuis le jour où il faut effectuer une modification dans *n* versions dupliquées. Ici on a placé dans le fichier *Connect.php* les informations de base sur la connexion à MySQL : le nom du serveur, le nom de la base et le compte d'accès à la base.

Exemple 1.7 *exemples/Connect.php* : *Fichier contenant les paramètres de connexion*

```
<?php
   //
   // Fichier contenant les definitions de constantes
   // pour la connexion à MySQL

   define ('NOM',"adminFilms");
   define ('PASSE', "mdpAdmin");
   define ('SERVEUR', "localhost");
   define ('BASE', "films");
?>
```

La commande `define` permet de définir des *constantes*, ou symboles correspondant à des valeurs qui ne peuvent être modifiées. L'utilisation systématique des constantes, définies en un seul endroit (un fichier que l'on peut insérer à la demande) garantit l'évolutivité du site. Si le serveur devient par exemple *magellan* et le nom de la base *Movies*, il suffira de faire la modification dans cet unique fichier. Accessoirement, l'utilisation de symboles simples permet de ne pas avoir à se souvenir de valeurs ou de textes qui peuvent être compliqués.

> **REMARQUE –** Il est tentant de donner une extension autre que *.php* aux fichiers contenant les scripts. Le fichier *Connect.php* par exemple pourrait être nommé *Connect.inc* pour bien indiquer qu'il est destiné à être inclus dans d'autres scripts. Attention cependant : il devient alors possible de consulter le contenu du fichier avec l'URL *http://serveur/Connect.inc*. L'extension *.inc* étant inconnue du programme serveur, ce dernier choisira de transmettre le contenu en clair (en-tête `text/plain`) au client. Cela serait très regrettable dans le cas de *Connect.php*, qui contient un mot de passe. Un fichier d'extension *.php* sera, lui, toujours soumis par le programme serveur au filtre de l'interpréteur PHP et son contenu n'est jamais visible par le client web.
>
> Il faut protéger le plus possible les fichiers contenant des mots de passe. L'accès à ces fichiers devrait être explicitement réservé aux utilisateurs qui doivent les modifier (le webmestre) et au serveur web (dans ce dernier cas, un accès en lecture suffit).

Connexion au serveur

Donc nous disposons avec ce `require` des symboles de constantes `NOM`, `PASSE`, `BASE` et `SERVEUR`[2], soit tous les paramètres nécessaires à la connexion à MySQL.

```
$connexion = mysql_pconnect (SERVEUR, NOM, PASSE);
```

La fonction `mysql_pconnect()` essaie d'établir une connexion avec le serveur *mysqld*. En cas de succès une valeur positive est renvoyée, qui doit ensuite être utilisée pour dialoguer avec le serveur. En cas d'échec `mysql_pconnect()` affiche un message d'erreur et renvoie une valeur nulle.

2. L'utilisation des majuscules pour les constantes n'est pas une obligation, mais facilite la lecture.

> **REMARQUE –** Si vous voulez éviter que MySQL envoie un message en cas d'échec à la connexion, vous pouvez préfixer le nom de la fonction par « @ ». C'est à vous alors de tester si la connexion est établie et d'afficher un message selon vos propres normes de présentation. Cette pratique est valable pour les autres fonctions de l'interface MySQL/PHP.

Avant de continuer, il faut vérifier que la connexion est bien établie. Pour cela, on peut tester la valeur de la variable `$connexion`, et, le cas échéant, afficher un message et interrompre le script avec `exit`.

```
if (!$connexion) {
  echo "Désolé, connexion à " . SERVEUR . " impossible\n";
  exit;
}
```

Avec PHP, toute valeur non nulle est considérée comme vraie, le 0 ou la chaîne vide étant interprétés comme faux. Au lieu d'effectuer un test de comparaison, on peut tester directement la valeur de la variable `$connexion`. Le test simple `if ($connexion)` donne un résultat inverse de `if ($connexion == 0)`.

En revanche, en inversant la valeur booléenne de `$connexion` avec l'opérateur de négation « ! », on obtient un test équivalent, et la notation, très courante, `if (! $connexion)`. La condition est vérifiée si `$connexion` est faux, ce qui est le but recherché.

Le même principe est appliqué au résultat de la fonction `mysql_select_db()` qui renvoie, elle aussi, une valeur positive (donc vraie) si l'accès à la base réussit. D'où le test :

```
if (!mysql_select_db (BASE, $connexion))
```

Tous ces tests sont importants. Beaucoup de raisons peuvent rendre un serveur indisponible, ou un compte de connexion invalide. Le fait de continuer le script, et donc d'effectuer des requêtes sans avoir de connexion, mène à des messages d'erreur assez désagréables. Bien entendu l'écriture systématique de tests et de messages alourdit le code : nous verrons comment écrire ce genre de chose une (seule) fois pour toutes en utilisant des fonctions.

Exécution de la requête

Le moment est enfin venu d'effectuer une requête ! On utilise la fonction `mysql_query()`.

```
$resultat = mysql_query ("SELECT * FROM FilmSimple", $connexion);
```

Comme d'habitude, cette fonction renvoie une valeur positive si la fonction s'exécute correctement. En cas de problème (erreur de syntaxe par exemple), le bloc associé au `else` est exécuté. Il affiche le message fourni par MySQL *via* la fonction `mysql_error()`.

```
echo "<b>Erreur dans l'exécution de la requête.</b><br/>";
echo "<b>Message de MySQL :</b> " .  mysql_error();
```

Noter l'utilisation de balises HTML dans les chaînes de caractères, ainsi que l'utilisation de l'opérateur de concaténation de chaînes, « . ».

Affichage du résultat

Si la requête réussit, il ne reste plus qu'à récupérer le résultat. Ici nous avons à résoudre un problème classique d'interaction entre une base de données et un langage de programmation. Le résultat est un ensemble, arbitrairement grand, de lignes dans une table, et le langage ne dispose pas de structure pratique pour représenter cet ensemble. On peut penser à tout mettre dans un tableau à deux dimensions (c'est d'ailleurs possible avec PHP), mais se pose alors un problème d'occupation mémoire si le résultat est *vraiment* volumineux (plusieurs mégaoctets par exemple).

La technique habituellement utilisée est de parcourir les lignes une à une avec un *curseur* et d'appliquer le traitement à chaque ligne individuellement. Cela évite d'avoir à charger tout le résultat en même temps. Ici, on utilise une des fonctions *fetch* qui correspondent à l'implantation de cette notion de curseur dans MySQL.

```
$film = mysql_fetch_object ($resultat);
```

La fonction `mysql_fetch_object()` prend une ligne dans le résultat (initialement on commence avec la première ligne) et positionne le curseur sur la ligne suivante. À chaque appel on progresse d'une étape dans le parcours du résultat. Quand toutes les lignes ont été parcourues, la fonction renvoie 0.

Avec cette fonction, chaque ligne est renvoyée sous la forme d'un *objet*, que nous référençons avec la variable `$film` dans l'exemple. Nous aurons l'occasion de revenir sur ce concept, pour l'instant il suffit de considérer qu'un objet est un groupe de valeurs, chacune étant identifiée par un nom.

Dans notre cas ces noms sont naturellement les noms des attributs de la table *FilmSimple*. On accède à chaque attribut avec l'opérateur '`->`'. Donc `$film->titre` est le titre du film, `$film->annee` l'année de réalisation, etc.

L'opération d'affectation du résultat de `mysql_fetch_object()` à la variable `$film` envoie elle-même une valeur, qui est 0 quand le résultat a été parcouru en totalité [3]. D'où la boucle d'affichage des films :

```
while ($film = mysql_fetch_object ($resultat)) {
  echo "$film->titre, paru en $film->annee, réalisé "
  . "par $film->prenom_realisateur $film->nom_realisateur.<br/>\
    n";
}
```

On peut remarquer, dans l'instruction `echo` ci-dessus, l'introduction de variables directement dans les chaînes de caractères. Autre remarque importante : on utilise deux commandes de retour à la ligne, <**br/**> et `\n`. Elles n'ont pas du tout la même fonction, et il est instructif de réfléchir au rôle de chacune.

- la balise <**br/**> indique au navigateur qu'un saut de ligne doit être effectué après la présentation de chaque film ;
- le caractère `\n` indique qu'un saut de ligne doit être effectué dans le *texte* HTML, pas dans la *présentation* du document qu'en fait le navigateur. Ce `\n`

3. Voir le chapitre 11 et la partie sur les expressions, page 426, pour plus d'explications.

n'a en fait aucun effet sur cette présentation puisque le format du texte HTML peut être quelconque. En revanche, il permet de rendre ce texte, produit automatiquement, plus clair à lire si on doit y rechercher une erreur.

Voici le texte HTML produit par le script, tel qu'on peut le consulter avec la commande *View source* du navigateur. Sans ce `\n`, tous les films seraient disposés sur une seule ligne.

Exemple 1.8 *exemples/ResMYPHP1.html* : *Résultat (texte HTML) du script*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Connexion à MySQL</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>Interrogation de la table FilmSimple</h1>

Alien, paru en 1979, réalisé par Ridley Scott.<br/>
Vertigo, paru en 1958, réalisé par Alfred Hitchcock.<br/>
Psychose, paru en 1960, réalisé par  Alfred Hitchcock.<br/>
Kagemusha, paru en 1980, réalisé par  Akira Kurosawa.<br/>
Volte-face, paru en 1997, réalisé par  John Woo.<br/>
Titanic, paru en 1997, réalisé par  James Cameron.<br/>
Sacrifice, paru en 1986, réalisé par  Andrei Tarkovski.<br/>

</body>
</html>
```

1.4.2 Formulaires d'interrogation

Une des forces de PHP est son intégration naturelle avec les formulaires HTML. Les valeurs saisies dans les champs du formulaire sont directement fournies dans le tableau `$_POST` ou `$_GET` selon le mode choisi, ainsi que dans le tableau `$_REQUEST` dans tous les cas. L'utilisation de SQL donne des commandes plus simples et plus puissantes.

Voici le formulaire d'interrogation :

Exemple 1.9 *exemples/ExForm3.html* : *Formulaire d'interrogation*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
```

```
<head>
<title>Formulaire pour script PHP/MySQL</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>Formulaire pour script PHP/MySQL</h1>

<form action="ExMyPHP2.php" method='get'>

Ce formulaire vous permet d'indiquer des paramétres pour
la recherche de films :
  <p>
  Titre : <input type='text' size='20' name = 'titre' value='%'/
      ><br/>
  Le caractère '%' remplace n'importe quelle chaîne.
  </p><p>
  Année début : <input type='text' size='4' name='anMin' value
      ='1900'/>
  Année fin : <input type='text' size='4' name='anMax' value
      ='2100'/> <br/>

  <b>Comment combiner ces critères. </b>
  ET <input type='radio' name='comb' value='ET' checked='1'/>
  OU <input type='radio' name='comb' value='OU'/> ?
  <p/>
  <input type='submit' value='Rechercher'/>
</form>

</body>
</html>
```

L'attribut `action` fait référence au script PHP à exécuter. On peut entrer dans le champ `titre` non seulement un titre de film complet, mais aussi des titres partiels, complétés par le caractère « % » qui signifie, pour SQL, une chaîne quelconque. Donc on peut, par exemple, rechercher tous les films commençant par « `Ver` » en entrant « `Ver%` », ou tous les films contenant un caractère blanc avec « `% %` ». Le fichier ci-dessous est le script PHP associé au formulaire précédent. Pour plus de concision, nous avons omis tous les tests portant sur la connexion et l'exécution de la requête SQL qui peuvent – devraient – être repris comme dans l'exemple 1.6, page 37.

Exemple 1.10 *exemples/ExMyPHP2.php* : *Le script associé au formulaire de l'exemple 1.9*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Résultat de l'interrogation</title>
```

```
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>Résultat de l'interrogation par formulaire</h1>

<?php
require ("Connect.php");

// Prenons les variables dans le tableau. C'est sûrement
// le bon endroit pour effectuer des contrôles.

$titre = $_GET['titre'];
$anMin = $_GET['anMin'];
$anMax = $_GET['anMax'];
$comb  = $_GET['comb'];

echo "<b>Titre = $titre anMin = $anMin anMax=$anMax\n";
echo "Combinaison logique : $comb</b><br/>\n";

// Créons la requête en tenant compte de la combinaison logique
if ($comb == 'ET')
  $requete = "SELECT * FROM FilmSimple "
    . "WHERE titre LIKE '$titre' "
    . "AND annee BETWEEN $anMin and $anMax";
 else
   $requete = "SELECT * FROM FilmSimple "
     . "WHERE titre LIKE '$titre' "
     . "OR (annee BETWEEN $anMin and $anMax)";

$connexion = mysql_pconnect (SERVEUR, NOM, PASSE);
mysql_select_db (BASE, $connexion);

// Exécution et affichage de la requête

$resultat = mysql_query ($requete, $connexion);

while ( ($film = mysql_fetch_object ($resultat)))
  echo "$film->titre, paru en $film->annee, réalisé "
    . "par $film->prenom_realisateur $film->nom_realisateur.<br
       />\n";
?>
</body>
</html>
```

Les variables `$titre`, `$anMin`, `$anMax` et `$comb` sont placées dans le tableau `$_GET` (noter que le formulaire transmet ses données en mode `get`). Pour clarifier le code, on les place dans des variables simples au début du script :

```
$titre = $_GET['titre'];
$anMin = $_GET['anMin'];
$anMax = $_GET['anMax'];
$comb  = $_GET['comb'];
```

En testant la valeur de `$comb`, qui peut être soit « `ET` », soit « `OU` », on détermine quel est l'ordre SQL à effectuer. Cet ordre utilise deux comparateurs, `LIKE` et `BETWEEN`. `LIKE` est un opérateur de *pattern matching* : il renvoie vrai si la variable `$titre` de PHP peut être rendue égale à l'attribut `titre` en remplaçant dans `$titre` le caractère '%' par n'importe quelle chaîne.

La requête SQL est placée dans une chaîne de caractères qui est ensuite exécutée.

```
$requete = "SELECT * FROM FilmSimple "
        . "WHERE titre LIKE '$titre' "
        . "AND annee BETWEEN $anMin and $anMax";
```

Dans l'instruction ci-dessus, on utilise la concaténation de chaînes (opérateur « . ») pour disposer de manière plus lisible les différentes parties de la requête. On exploite ensuite la capacité de PHP à reconnaître l'insertion de variables dans une chaîne (grâce au préfixe $) et à remplacer ces variables par leur valeur. En supposant que l'on a saisi `Vertigo`, 1980 et 2000 dans ces trois champs, la variable `$requete` sera la chaîne suivante :

```
SELECT * FROM FilmSimple
WHERE titre LIKE 'Vertigo'
AND annee BETWEEN 1980 AND 2000
```

Il faut toujours encadrer une chaîne de caractères comme `$titre` par des apostrophes simples « ' » car MySQL ne saurait pas faire la différence entre `Vertigo` et le nom d'un attribut de la table. De plus cette chaîne de caractères peut éventuellement contenir des blancs, ce qui poserait des problèmes. Les apostrophes simples sont acceptées au sein d'une chaîne encadrée par des apostrophes doubles, et réciproquement.

> **REMARQUE –** Que se passe-t-il si le titre du film contient lui même des apostrophes simples, comme, par exemple, « `L'affiche rouge` » ? Et bien il faut préfixer par « \ », avant la transmission à MySQL, tous les caractères qui peuvent être interprétés comme des délimiteurs de chaîne (et plus généralement tous les caractères spéciaux). La chaîne transmise sera donc `L\'affiche rouge`, et MySQL interprétera correctement cet apostrophe comme faisant partie de la chaîne et pas comme un délimiteur.
>
> Ce comportement de PHP est activé par l'option `magic_quotes_gpc` qui se trouve dans le fichier de configuration *php.ini*. Cette option tend à être à `Off` dans les versions récentes de PHP, et il faut alors recourir aux fonctions `addSlashes()` (ou, mieux, `mysql_escape_string()`) et `stripSlashes()` qui permettent d'ajouter ou des supprimer les caractères d'échappement dans une chaîne de caractères. Nous reviendrons longuement sur cet aspect délicat dans le prochain chapitre.

En revanche, les apostrophes sont inutiles pour les valeurs numériques comme `$anMin` et `$anMax` qui ne peuvent être confondus avec des noms d'attribut et ne soulèvent donc pas de problème d'interprétation. Il faut quand même noter que nous ne faisons aucun contrôle sur les valeurs saisies, et qu'un utilisateur malicieux qui place des caractères alphanumériques dans les dates, ou transmet des chaînes vides, causera quelques soucis à ce script (vous pouvez d'ailleurs essayer, sur notre site).

Une dernière remarque : ce script PHP est associé au formulaire de l'exemple 1.9 puisqu'il attend des paramètres que le formulaire a justement pour objectif de collecter et transmettre. Cette association est cependant assez souple pour que tout autre moyen de passer des paramètres (dans le bon mode, `get` ou `post`) au script soit acceptée. Par exemple l'introduction des valeurs dans l'URL, sous la forme ci-dessous, est tout à fait valable puisque les variables sont attendues en mode `get`.

```
ExMyPHP2.php?titre=Vert%&anMin=1980&anMax=2000&comb=OR
```

Pour tout script, on peut donc envisager de se passer du formulaire, soit en utilisant la méthode ci-dessus si la méthode est `get`, soit en développant son propre formulaire ou tout autre moyen de transmettre les données en mode `post`. Il est possible par exemple de récupérer la description (sous forme HTML) du film *Vertigo* avec l'URL *http://us.imdb.com/Title?Vertigo*, qui fait directement appel à un programme web du site *imdb.com*. Cela signifie en pratique que l'on n'a pas de garantie sur la provenance des données soumises à un script, et qu'elles n'ont pas forcément été soumises aux contrôles (JavaScript ou autres) du formulaire prévu pour être associé à ce script. Des vérifications au niveau du serveur s'imposent, même si nous les omettons souvent dans ce livre pour ne pas alourdir nos exemples.

1.4.3 Formulaires de mises à jour

L'interaction avec un site comprenant une base de données implique la possibilité d'effectuer des mises à jour sur cette base. Un exemple très courant est *l'inscription* d'un visiteur afin de lui accorder un droit d'utilisation du site. Là encore les formulaires constituent la méthode normale de saisie des valeurs à placer dans la base.

Nous donnons ci-dessous l'exemple de la combinaison d'un formulaire et d'un script PHP pour effectuer des insertions, modifications ou destructions dans la base de données des films. Cet exemple est l'occasion d'étudier quelques techniques plus avancées de définition de tables avec MySQL et de compléter le passage des paramètres entre le formulaire et PHP.

Une table plus complète

L'exemple utilise une version plus complète de la table stockant les films.

Exemple 1.11 *exemples/FilmComplet.sql* : *Fichier de création de FilmComplet*

```
/* Création d'une table FilmComplet    */

CREATE TABLE FilmComplet
  (titre          VARCHAR (30),
   annee          INTEGER,
   nom_realisateur  VARCHAR (30),
   prenom_realisateur  VARCHAR (30),
   annee_naissance INTEGER,
   pays    ENUM ("FR", "US", "DE", "JP"),
```

```
    genre   SET ("C", "D", "H", "S"),
    resume          TEXT
  )
;
```

La table *FilmComplet* comprend quelques nouveaux attributs, dont trois utilisent des types de données particuliers.

1. l'attribut `pays` est un type *énuméré* : la valeur – unique – que peut prendre cet attribut doit appartenir à un ensemble donné explicitement au moment de la création de la table avec le type `ENUM` ;
2. l'attribut `genre` est un type *ensemble* : il peut prendre une *ou plusieurs* valeurs parmi celle qui sont énumérées avec le type `SET` ;
3. enfin l'attribut `resume` est une *longue* chaîne de caractères de type `TEXT` dont la taille peut aller jusqu'à 65 535 caractères (soit $2^{16} - 1$: la longueur de la chaîne est codée sur 2 octets = 16 bits).

Ces trois types de données ne font pas partie de la norme SQL. En particulier, une des règles de base dans un SGBD relationnel est qu'un attribut, pour une ligne donnée, ne peut prendre plus d'une seule valeur. Le type `SET` de MySQL permet de s'affranchir – partiellement – de cette contrainte. On a donc décidé ici qu'un film pouvait appartenir à plusieurs genres.

Le formulaire

Le formulaire permettant d'effectuer des mises à jour sur la base (sur la table *FilmComplet*) est donné ci-dessous.

Exemple 1.12 *exemples/ExForm4.html* : *Formulaire de mise à jour*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Formulaire complet</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

 <form  action="ExMyPHP3.php"  method='post'>

  Titre : <input type='text' size='20' name="titre"/><br/>
  Année : <input type='text' size='4' maxlength='4'
              name="annee" value="2000"/>
  <p>
  Comédie    : <input type='checkbox' name='genre[]' value='C'/>
  Drame      : <input type='checkbox' name='genre[]' value='D'/>
```

```
  Histoire   : <input type='checkbox' name='genre[]' value='H'/>
  Suspense   : <input type='checkbox' name='genre[]' value='S'/>
  </p><p>
   France       : <input type='radio' name='pays' value='FR'
       checked='1'/>
   Etats-Unis : <input type='radio' name='pays' value='US'/>
   Allemagne  : <input type='radio' name='pays' value='DE'/>
   Japon      : <input type='radio' name='pays' value='JP'/>
  </p>
  <p>
  Metteur en scène (prénom - nom) :
    <input type='text' size='20' name="prenom"/>
    <input type='text' size='20' name="nom"> <br/>

  Année de naissance : <input type='text' size='4' maxlength='4'
             name="annee_naissance" value='2000'/>
  </p>
  Résumé :  <textarea name='resume' cols='30' rows='3'>Résumé du
      film
            </textarea>

  <h1>Votre choix</h1>
  <input type='submit' value='Insérer' name='inserer'/>
  <input type='submit' value='Modifier' name='modifier'/>
  <input type='submit' value='Détruire' name='detruire'/>
  <input type='reset' value='Annuler'/>
</form>

</body>
</html>
```

Il est assez proche de celui de l'exemple 1.2, page 13, avec quelques différences notables. Tout d'abord, le nom du champ `genre` est `genre[]`.

```
Comédie    : <input type='checkbox' name='genre[]' value='C'/>
Drame      : <input type='checkbox' name='genre[]' value='D'/>
Histoire   : <input type='checkbox' name='genre[]' value='H'/>
Suspense   : <input type='checkbox' name='genre[]' value='S'/>
```

Pour comprendre l'utilité de cette notation, il faut se souvenir que les paramètres issus du formulaire sont passés au script sur le serveur sous la forme de paires *`nom=valeur`*. Ici on utilise un champ `checkbox` puisqu'on peut affecter plusieurs genres à un film. Si on clique sur au moins deux des valeurs proposées, par exemple « Histoire » et « Suspense », la chaîne transmise au serveur aura la forme suivante :

```
...&genre[]=H&genre[]=S&...
```

Pour le script PHP exécuté par le serveur, cela correspond aux deux instructions suivantes :

```
$genre[] = 'H';
$genre[] = 'S';
```

Imaginons un instant que l'on utilise un nom de variable `$genre`, sans les crochets []. Alors pour PHP la deuxième affectation viendrait annuler la première et `$genre` n'aurait qu'une seule valeur, 'S'. La notation avec crochets indique que `$genre` est en fait un *tableau*, donc une liste de valeurs. Mieux : PHP incrémente automatiquement l'indice pour chaque nouvelle valeur placée dans un tableau. Les deux instructions ci-dessus créent un tableau avec deux entrées, indicées respectivement par 0 et 1, et stockant les deux valeurs 'H' et 'S'.

Une autre particularité du formulaire est l'utilisation de plusieurs boutons `submit`, chacun associé à un nom différent.

```
 <input type='submit' value='Insérer' name='inserer'/>
<input type='submit' value='Modifier' name='modifier'/>
<input type='submit' value='Détruire' name='detruire'/>
```

Quand l'utilisateur clique sur l'un des boutons, une seule variable PHP est créée, dont le nom correspond à celui du bouton utilisé. Le script peut tirer parti de ce mécanisme pour déterminer le type d'action à effectuer.

Le script PHP

La troisième composante de cette application de mise à jour est le script PHP.

Exemple 1.13 *exemples/ExMyPHP3.php* : *Le script de mise à jour de FilmComplet*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Résultat de la mise à jour</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>Résultat de la mise à jour par formulaire</h1>

<?php
require ("Connect.php");

// Récupération des variables.
// Quelques contrôles seraient nécessaires...

$titre  = $_POST['titre'];
$annee  = $_POST['annee'];
$pays   = $_POST['pays'];
$prenom = $_POST['prenom'];
$nom    = $_POST['nom'];
$annee_naissance = $_POST['annee_naissance'];
$resume = $_POST['resume'];
```

```
// Il peut n'y avoir aucun genre saisi...
if (!isSet($_POST['genre']))
     $genre=array();
else
     $genre  = $_POST['genre'];

echo "<hr/><h2>\n";

// Test du type de la mise à jour effectuée

if (isSet($_POST['inserer']))
  echo "Insertion du film $titre";
else if (isSet($_POST['modifier']))
   echo "Modification du film $titre";
else if (isSet($_POST['detruire']))
  echo "Destruction du film $titre";

echo "</h2><hr/>\n";

// Affichage des données du formulaire

echo "Titre: $titre <br/> annee: $annee<br/>Pays: $pays<br/>\n";

// Préparation de la chaîne pour insérer
$chaine_genre = ""; $separateur = "";
for ($i=0; $i < count ($genre); $i++) {
  $chaine_genre .= $separateur . $genre[$i];
  $separateur = ",";
}

echo "Genres  = $chaine_genre<br/>";
echo "Résumé  = $resume<br/>\n";
echo "Mis en scène par $prenom $nom\n";

//  Connexion à la base, et création de l'ordre SQL

$connexion = mysql_pconnect (SERVEUR, NOM, PASSE);
mysql_select_db (BASE, $connexion);

if (isSet($_POST['inserer']))
  $requete = "INSERT INTO FilmComplet (titre, annee, "
    . "prenom_realisateur, nom_realisateur, annee_naissance,  "
    . "pays, genre, resume) VALUES ('$titre', $annee, "
    . "'$prenom', '$nom', $annee_naissance, "
    . " '$pays', '$chaine_genre', '$resume') ";
if (isSet($_POST['modifier']))
  $requete = "UPDATE FilmComplet SET annee=$annee, "
    . "prenom_realisateur = '$prenom', nom_realisateur='$nom', "
    . "annee_naissance=$annee_naissance, pays='$pays', "
    . "genre = '$chaine_genre', resume='$resume' "
    . " WHERE titre = '$titre' ";
```

```
if (isSet($_POST['detruire']))
  $requete = "DELETE FROM FilmComplet WHERE  titre = '$titre'";

// Exécution de l'ordre SQL (un test d'erreur serait bienvenu)

$resultat = mysql_query ($requete, $connexion);
if ($resultat)
  echo "<hr/>La requête '$requete' a été effectuée.\n";
else {
  echo "La requête n'a pu être exécutée pour la raison suivante:"
    . mysql_error($connexion);
}
?>
</body>
</html>
```

Ce script procède en plusieurs étapes, chacune donnant lieu à une insertion dans la page HTML fournie en retour au serveur.

Tout d'abord, on récupère les paramètres transmis *en principe* par le formulaire. En pratique rien ne garantit, encore une fois, qu'un utilisateur malicieux ne va pas appeler le script sans utiliser le formulaire et sans même passer un paramètre. Il faudrait tester l'existence des paramètres attendus, si la sécurité était importante. Ce test peut être effectué avec la fonction `isSet()`, et un exemple est ici donné pour le paramètre `genre` :

```
// Il peut n'y avoir aucun genre saisi...
if (!isSet($_POST['genre']))
     $genre=array();
else
     $genre  = $_POST['genre'];
```

Si on constate qu'aucun genre n'est transmis (ce qui peut arriver même si l'on utilise le formulaire puisque ce dernier ne comprend pas de contrôles), on initialise la variable `$genre` avec un tableau vide (`array()`). Ce type de contrôle pourrait/devrait être effectué pour tous les paramètres : c'est fastidieux mais souvenez-vous qu'un script est un programme en accès libre pour le monde entier...

On contrôle ensuite le bouton de déclenchement utilisé. Selon le cas, on trouve un élément `'inserer'`, `'modifier'`, ou `'detruire'` dans le tableau `$_POST`, et on en déduit le type de mise à jour effectué. On l'affiche alors pour informer l'utilisateur que sa demande a été prise en compte. On utilise encore la fonction `isSet()` de PHP pour tester l'existence d'une variable (ici une entrée dans un tableau).

```
if (isSet($_POST['inserer']))
      echo "Insertion du film $titre";
elseif (isSet($_POST['modifier']))
      echo "Modification du film $titre";
elseif (isSet($_POST['detruire']))
      echo "Destruction du film $titre";
```

La construction `if-elseif` permet de contrôler successivement les différentes valeurs possibles. On pourrait aussi utiliser une structure `switch`, ce qui permettrait en outre de réagir au cas où aucune des variables ci-dessus n'est définie.

On récupère ensuite les valeurs provenant du formulaire et on les affiche. La variable `$genre` est traitée de manière particulière.

```
$chaine_genre = ""; $separateur = "";
for ($i=0; $i < count ($genre); $i++) {
    $chaine_genre .= $separateur . $genre[$i];
    $separateur = ",";
}
echo "Genres  = $chaine_genre<br/>";
```

Notez l'initialisation des variables `$chaine_genre` et `$separateur`. PHP peut parfois se montrer assez laxiste et accepter l'utilisation de variables non déclarées, en leur donnant alors la valeur 0 ou la chaîne vide selon le contexte. On peut envisager d'en tirer parti, mais dans certaines configurations – de plus en plus courantes – le niveau de contrôle (défini par l'option `error_reporting` dans le fichier de configuration) est très élevé et ce genre de pratique engendre des messages d'avertissement très désagréables. Mieux vaut donc prendre l'habitude d'initialiser les variables.

Rappelons que `$genre` est un tableau, dont chaque élément correspond à un des choix de l'utilisateur. La fonction `count()` permet de connaître le nombre d'éléments, puis la boucle `for` est utilisée pour parcourir un à un ces éléments.

Au passage, on crée la variable `$chaine_genre`, une chaîne de caractères qui contient la liste des codes de genres, séparés par des virgules, selon le format attendu par MySQL. Si, par exemple, on a choisi « Histoire » et « Suspense » `$chaine_genre` contiendra "H,S".

Enfin on construit la requête `INSERT`, `UPDATE` ou `DELETE` selon le cas.

Discussion

Le script précédent a beaucoup de défauts qui le rendent impropre à une véritable utilisation. Une première catégorie de problèmes découle de la conception de la base de données elle-même. Il est par exemple possible d'insérer plusieurs fois le même film, une mise à jour peut affecter plusieurs films, il faut indiquer à chaque saisie l'année de naissance du metteur en scène même s'il figure déjà dans la base, etc. Nous décrirons dans le chapitre 4 une conception plus rigoureuse qui permet d'éviter ces problèmes.

Si on se limite à la combinaison HTML/PHP en laissant pour l'instant de côté la base MySQL, les faiblesses du script sont de deux natures.

Pas de contrôles. Aucun test n'est effectué sur les valeurs des données, et en particulier des chaînes vides peuvent être transmises pour tous les champs. De plus, la connexion à MySQL et l'exécution des requêtes peuvent échouer pour des quantités de raisons : cet échec éventuel devrait être contrôlé.

Une ergonomie rudimentaire. En se restreignant à des scripts très simples, on a limité du même coup la qualité de l'interface. Par exemple, on aimerait que les formulaires soient présentés avec un alignement correct des champs. Plus important : il serait souhaitable, au moment de mettre à jour les informations d'un film, de disposer comme valeur par défaut des valeurs déjà saisies

Ces problèmes peuvent se résoudre en ajoutant du code PHP pour effectuer des contrôles, ou en alourdissant la partie HTML. Le risque est alors d'aboutir à des scripts illisibles et difficilement maintenables. Le chapitre qui suit va montrer comment mettre en œuvre des scripts plus robustes et offrant une interface utilisateur plus sûre et plus conviviale. Nous verrons également comment éviter des scripts très longs et difficilement lisibles en structurant le code de manière à répartir les tâches en unités indépendantes.

2

Techniques de base

Ce chapitre montre, sans entrer trop rapidement dans des techniques de programmation avancées (bibliothèques de fonctions, programmation objet, conception d'une base de données) comment réaliser les bases d'un site avec MySQL et PHP. Le but est double : donner un catalogue des fonctionnalités les plus courantes, et montrer à cette occasion les principales techniques de gestion des interactions web client/serveur. Voici les sujets abordés :

1. **Réutilisation de code**. Dès que l'on commence à produire les pages d'un site à partir de scripts PHP communiquant avec MySQL, on est amené à programmer de manière répétitive des parties de code correspondant soit à des opérations routinières (connexion à la base, exécution d'une requête), soit à des tests (validations des champs de saisie, vérification que des instructions se sont exécutées correctement), soit enfin à du texte HTML. Les fonctions constituent un des principaux moyens (l'autre étant la programmation orientée-objet, présentée dans le chapitre suivant) de réutiliser le code en évitant par conséquent de répéter indéfiniment les mêmes instructions (page 56).
2. **Traitement des données transmises par HTTP**. Une des principales spécificités de la programmation web est la réception et l'envoi de données via le protocole HTTP. PHP simplifie considérablement le traitement de ces données, et permet souvent de les manipuler sans tenir compte du protocole, comme s'il s'agissait de paramètres passés au script. Dans de nombreux cas, il faut cependant être attentif aux transformations subies par les données, soit parce qu'elle doivent être codées conformément au protocole HTTP, soit parce que le décryptage de PHP introduit des modifications parfois non souhaitées, soit enfin à cause de failles potentielles de sécurité. Cela soulève des problèmes délicats liés aux différentes configurations possibles de PHP et aux différents contextes d'utilisation (HTML, MySQL, texte

simple, ...). Cette section discute cet aspect essentiel et décrit une stratégie générale pour régler ces problèmes (page 64).

3. **Gestion des formulaires**. La fin du chapitre précédent a montré comment mettre à jour une table à partir d'un formulaire. Nous reprenons le sujet de manière beaucoup plus détaillée, en montrant comment contrôler les saisies, comment réafficher le formulaire avec des valeurs par défaut, comment utiliser un seul formulaire pour insertions et mises à jour (page 78).
4. **Transfert et gestion de fichiers**. Un site web n'est pas seulement un ensemble de pages HTML. Il peut également fournir ou recevoir des fichiers. Nous montrons comment transmettre des fichiers du client au serveur, comment les stocker et référencer sur ce dernier, et comment les présenter pour téléchargement. Ces fonctionnalités sont présentées dans le cadre de la gestion d'un petit album photo (page 90).
5. **Gestion des sessions**. Le protocole HTTP ne mémorise pas les échanges entre un client et un serveur. Pour effectuer un suivi, on doit donc simuler des « sessions », le plus souvent à l'aide de *cookies*. Nous détaillons pas à pas la réalisation d'une session (page 98).
6. **SQL dynamique**. Cette section montre comment traiter des requêtes SQL déterminées « dynamiquement » à l'exécution d'un script, et comment afficher le résultat – qui peut être arbitrairement grand – en plusieurs pages, à la manière d'un moteur de recherche (page 109).

Tous ces sujets sont traités indépendamment pour permettre au lecteur de s'y reporter directement après une première lecture, et s'appuient sur des exemples complets, utilisables et modifiables. Les techniques présentées dans ce chapitre forment les briques de base pour la construction d'un site complet, sujet abordé après l'étude de la programmation objet, dans le prochain chapitre. Nous verrons alors comment intégrer ces techniques dans une démarche globale, comprenant une conception rigoureuse de la base de données, le choix d'un style de développement cohérent et la séparation des codes HTML et PHP.

2.1 PROGRAMMATION AVEC FONCTIONS

Une fonction est une partie de code qui ne peut communiquer avec le script appelant que par l'intermédiaire d'un petit nombre de variables – les paramètres – bien identifiées. Toutes les données utilisées localement par la fonction pour accomplir sa tâche particulière ne sont pas accessibles au script appelant. Réciproquement, la fonction ne peut pas manipuler les informations du script.

2.1.1 Création de fonctions

Un script reposent sur des fonctions confie à chacune l'implantation d'un service précisément identifié : ouvrir un fichier, lire une donnée, effectuer un calcul, etc. Chaque fonction accomplit son rôle (et rien de plus), et le script n'est alors rien

d'autre qu'un coordinateur qui délègue les diverses tâches à des fonctions, lesquelles à leur tour subdivisent leur travail en faisant appel à des fonctions plus spécialisées, et ainsi de suite. La structuration judicieuse d'un programme en fonctions concourt à la production d'un code sain, lisible et donc facile à mettre à jour. La conception de cette structuration vise à deux buts principaux :

1. *déléguer* les tâches ingrates, les données secondaires, les contrôles d'erreur à des modules particuliers ;
2. *partager* le code : idéalement, on ne devrait *jamais* écrire deux fois la même instruction car cette instruction devrait être implantée par une fonction appelée partout où l'on en a besoin.

En appliquant ces idées, on obtient un code dans lequel chaque fonction occupe au plus quelques dizaines de lignes dans un fichier à part du script principal, ce qui permet de comprendre facilement l'objectif poursuivi et l'algorithme mis en œuvre. *A contrario*, le mauvais script est celui qui cumule toutes les instructions dans un seul fichier : on aboutit rapidement à des centaines de lignes accumulant les structures principales et les détails les plus anodins dans un même code, rapidement illisible. Ce style de programmation (très courant) est à terme impossible à maintenir et corriger. Une fonction bien conçue, bien écrite et bien testée, c'est un problème sur lequel on ne reviendra plus jamais !

Écriture de fonctions

Nous allons commencer par définir une fonction `Connexion()` qui se connecte à MySQL. On peut se demander pourquoi définir une telle fonction, alors qu'il en existe déjà une : la réponse est tout simplement que `mysql_pconnect()` peut échouer pour diverses raisons, et renvoie alors une valeur nulle qui ne peut être utilisée pour exécuter des requêtes. Si l'on ne teste pas ce cas, on s'expose un jour ou l'autre à de gros ennuis. Le tester à chaque appel à `mysql_pconnect()` rompt le principe exposé ci-dessus de ne jamais écrire des instructions redondantes.

La fonction `Connexion()`

La fonction `Connexion()` prend comme paramètres les valeurs nécessaires pour se connecter à un serveur sous un compte utilisateur, et se placer ensuite dans une base si la connexion a réussi. Elle est placée dans un fichier *Connexion.php* qui peut être inclus avec la commande `require` dans n'importe quel script.

Exemple 2.1 *exemples/Connexion.php* : *Fonction de connexion à MySQL*

```
<?php

 // Fonction Connexion: connexion à MySQL

 function Connexion ($pNom, $pMotPasse, $pBase, $pServeur)
 {
  // Connexion au serveur
```

```
  $connexion = mysql_pconnect ($pServeur, $pNom, $pMotPasse);

  if (!$connexion) {
    echo "Désolé, connexion au serveur $pServeur impossible\n";
    exit;
  }

  // Connexion à la base
  if (!mysql_select_db ($pBase, $connexion)) {
    echo "Désolé, accès à la base $pBase impossible\n";
    echo "<b>Message de MySQL :</b> " . mysql_error($connexion);
    exit;
  }

  // On renvoie la variable de connexion
  return $connexion;
 } // Fin de la fonction
?>
```

La première ligne de la fonction est sa *signature* (ou prototype). Elle définit les paramètres que la fonction accepte. L'interpréteur vérifie, au moment de l'appel à une fonction, que le nombre de paramètres transmis correspond à celui de la signature.

L'apport essentiel de `Connexion()` par rapport à `mysql_pconnect()` est de tester le cas de l'échec de l'accès au serveur de MySQL et de prendre les mesures en conséquence. Les deux avantages de l'utilisation des fonctions donnés ci-dessus apparaissent dès cette simple implantation :

1. *délégation* : le script qui se connecte à MySQL a certainement des choses plus importantes à faire que de tester ce genre d'erreur ;
2. *partage* : c'est le bénéfice le plus apparent ici. On n'aura plus jamais à se soucier de l'échec de l'accès au serveur. De plus, la politique appliquée en cas d'échec est définie en un seul endroit. Ici on a choisi de quitter le script, mais le jour où l'on décide de créer un fichier dans *tmp* avec toutes les erreurs rencontrées, la modification affecte seulement la fonction `Connexion()`.

> **REMARQUE –** Pour l'instant les messages d'erreur sont affichés à l'écran. Sur un site en production c'est une très mauvaise pratique, pour des raisons d'image et de sécurité. La bonne méthode (quoique légèrement hypocrite) consiste à afficher un message courtois disant que le site est en maintenance, et à envoyer un message à l'administrateur pour lui signaler le problème.

Exécution de requêtes

Selon le même principe, il est possible de définir des fonctions pour exécuter une requête avec MySQL. Le fichier *ExecRequete.php* contient trois fonctions : la première pour exécuter une requête, la seconde et la troisième pour récupérer une ligne du résultat respectivement sous forme d'objet (un groupe `$o`, dont chaque valeur `v` est accessible par `$o->v`) ou de tableau associatif (un tableau `$t` dont chaque valeur `v` est accessible par `$t['v']`).

Exemple 2.2 *exemples/ExecRequete.php* : *Fonctions exécutant une requête*

```
<?php
 // Exécution d'une requête avec MySQL

 function ExecRequete ($requete, $connexion)
 {
  $resultat = mysql_query ($requete, $connexion);

  if ($resultat)
   return $resultat;
  else {
    echo "<b>Erreur dans l'exécution de la requête '$requete'.
          </b><br/>";
    echo "<b>Message de MySQL :</b> " .  mysql_error($connexion);
    exit;
  }
 } // Fin de la fonction ExecRequete

 // Recherche de l'objet suivant
 function ObjetSuivant ($resultat)
 {
   return  mysql_fetch_object ($resultat);
 }

 // Recherche de la ligne suivante (retourne un tableau)
 function LigneSuivante ($resultat)
 {
   return  mysql_fetch_assoc ($resultat);
 }
?>
```

Le regroupement de fonctions concourant à un même objectif – ici l'exécution d'une requête, puis l'exploitation du résultat – est une pratique classique et mène à la notion, elle aussi classique, de *module*. Un module est un ensemble de fonctionnalités qui correspond à une partie cohérente et bien identifiée d'une application plus large, placées en général dans un même fichier.

2.1.2 Utilisation des fonctions

Les définitions de fonctions doivent être placées dans des fichiers séparés, lesquels sont inclus avec l'instruction `require()` ou `require_once()` au début de chaque script qui fait appel à elles. Voici l'exemple 1.6, page 37, qui donnait un premier exemple d'accès à la base MySQL à partir d'un script PHP, maintenant réécrit avec quelques-unes des fonctions précédentes.

Exemple 2.3 *exemples/ExMyPHP4.php* : *L'exemple 1.6, avec des fonctions*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Connexion à MySQL</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>


<h1>Interrogation de la table FilmSimple</h1>

<?php
  require_once("Connect.php");
  require_once("Connexion.php");
  require_once("ExecRequete.php");

  $connexion = Connexion(NOM, PASSE, BASE, SERVEUR);
  $resultat = ExecRequete("SELECT * FROM FilmSimple", $connexion);

  while ($film = ObjetSuivant($resultat))
      echo "<b>$film->titre</b>, paru en $film->annee, réalisé "
   ."par $film->prenom_realisateur $film->nom_realisateur.<br/>\n";
?>
</body>
</html>
```

On peut apprécier l'économie réalisée dans la taille du code et la lisibilité qui en résulte. Entre autres avantages, il faut noter qu'il n'y a plus dans ce script aucune référence à MySQL. Le jour où l'on choisit d'utiliser – par exemple – PostgreSQL, les modifications ne touchent que les fonctions d'accès à la base et restent transparentes pour les autres scripts. La portabilité d'un code MySQL/PHP sur plusieurs SGBD sera développée dans le chapitre 3.

2.1.3 À propos de `require` et `include`

Il existe deux instructions pour inclure du code dans un fichier, `require` (et sa variante `require_once`) et `include`. La différence est subtile :

- `require(`*`fichier`*`)` se contente d'inclure le code de *`fichier`* dans le script courant, et tout se passe ensuite comme si l'instruction `require` avait été définitivement remplacée par le contenu de *`fichier`* ;
- `include(`*`fichier`*`)`, en revanche, correspond à une inclusion répétitive de *`fichier`*, chaque fois que l'instruction est rencontrée.

En général, c'est une mauvaise pratique que de placer des instructions dans un fichier pour l'exécuter avec `require` ou `include`. Le danger vient du fait que les variables manipulées dans les fichiers inclus viennent se confondre avec celles du script principal, avec des résultats imprévisibles et surtout difficilement détectables.

L'utilisation de fonctions est bien préférable car les variables des fonctions sont locales, et l'interaction avec le script se limite aux arguments de la fonction, facilement identifiables.

En conclusion, il est fortement recommandé d'utiliser seulement `require`, et de ne placer dans les fichiers inclus que des *définitions* de fonctions ou de constantes. On est sûr alors que le script ne contient ni variables ni instructions cachées. La fonction `include()` devrait être réservée aux cas où il faut déterminer, *à l'exécution*, le fichier à inclure. Un exemple possible est un site multi-langues dans lequel on crée un fichier pour chaque langue gérée.

Il est possible d'utiliser `require` récursivement. Voici par exemple le fichier *UtilBD.php* que nous utiliserons par la suite pour inclure en une seule fois les déclarations de constantes et de fonctions pour l'accès à MySQL.

Exemple 2.4 *exemples/UtilBD.php* : *Un fichier global d'inclusion des constantes et fonctions*

```
<?php
  // Fonctions et déclarations pour l'accès à MySQL
  require_once ("Connect.php");
  require_once ("Connexion.php");
  require_once ("ExecRequete.php");
?>
```

La variante `require_once` assure qu'un fichier n'est pas inclus deux fois dans un script (ce qui peut arriver lors d'inclusion transitives, un fichier qui en inclut un autre qui en inclut un troisième ...).

2.1.4 Passage par valeur et passage par référence

Une fonction prend en entrée des paramètres et renvoie une valeur qui peut alors être stockée dans une variable du script appelant, ou transmise comme paramètre à une autre fonction. Les paramètres sont passés *par valeur* en PHP. En d'autres termes, le programme appelant et la fonction disposent chacun d'un espace de stockage pour les valeurs de paramètres, et l'appel de la fonction déclenche la *copie* des valeurs depuis l'espace de stockage du programme appelant vers l'espace de stockage de la fonction.

REMARQUE – Attention, les objets sont passés par référence depuis la version 5 de PHP.

La conséquence essentielle est qu'une fonction *ne peut pas* modifier les variables du programme appelant puisqu'elle n'a pas accès à l'espace de stockage de ce dernier. Une fonction effectue une ou plusieurs opérations, renvoie éventuellement le résultat, mais ne modifie pas ses paramètres. Il s'agit d'une caractéristique importante (« pas d'effet de bord ») pour la lisibilité et la robustesse d'un programme. Elle permet en effet d'estimer avec certitude, en regardant un script constitué d'appels de fonctions, quel est l'effet de chacune. Ce n'est malheureusement plus vrai dès que l'on recours à des pratiques comme l'utilisation de variables globales et le passage par référence.

N'utilisez pas de variables globales si vous voulez garder un code sain. Quant au passage des paramètres par référence, il est possible en PHP. La notion de *référence*

(identique à celle du C++) correspond à la possibité de désigner un même contenu par plusieurs variables. Soit par exemple le fragment suivant

```
$a = 3;
$b = &$a;
```

Les variables `$a` et `$b` référencent alors le même contenu, dont la valeur est pour l'instant 3. Toute modification du contenu par l'intermédiaire de `$a` sera visible de `$b` et réciproquement. Par exemple :

```
$a = 5;
echo $b; // Affiche la valeur 5
```

Il faut souligner, pour ceux qui sont familiers avec le langage C, que les références *ne sont pas* des pointeurs puisque, contrairement à ces derniers, une référence est un symbole désignant un contenu pré-existant (celui d'une autre variable). Pour la question qui nous intéresse ici, elles peuvent cependant servir, comme les pointeurs C, à permettre le partage entre un contenu manipulé par un script et ce même contenu manipulé par une fonction. En passant en effet à une fonction la référence *r* à une variable *v* du script appelant, toute modification effectuée par la fonction sur *r* impactera *v*.

Il est tentant d'utiliser le passage par référence dans (au moins) les deux cas suivants :

1. pour des fonctions qui doivent renvoyer plusieurs valeurs ;
2. quand les paramètres à échanger sont volumineux et qu'on craint un impact négatif sur les performances.

En ce qui concerne le premier point, on peut remplacer le passage par référence par le renvoi de valeurs complexes, tableaux ou objets. Voici un exemple comparant les deux approches. Le premier est une fonction qui prend des références sur les variables du script principal et leur affecte le jour, le mois et l'année courante. Les variables `ceJour`, `ceMois` et `cetteAnnee` sont des références, et permettent donc d'accéder au même contenu que les variables du script appelant la fonction. Notez que le passage par référence est obtenu dans la *déclaration* de la fonction en préfixant par `&` le nom des paramètres.

Exemple 2.5 *exemples/References.php* : *Fonction avec passage par référence.*

```
<?php
// Exemple de fonction renvoyant plusieurs valeurs grâce à un
// passage par références

function aujourdhui_ref (&$ceJour, &$ceMois, &$cetteAnnee)
{
  // On calcule le jour, le mois et l'année courante
  $ceJour = date('d');
  $ceMois = date('m');
```

```
  $cetteAnnee = date('Y');

  // Rien à renvoyer !
}
?>
```

Voici maintenant la fonction équivalente renvoyant la date courante sous forme d'un tableau à trois entrées, `jour`, `mois` et `an`.

Exemple 2.6 *exemples/RenvoiTableau.php* : *Fonction renvoyant la date courante*

```
<?php
// Exemple de fonction renvoyant plusieurs valeurs grâce à un
// tableau

function aujourdhui_tab ()
{
  // Initialisation du retour
  $retour = array();

  // On calcule le jour, le mois et l'année courante
  $retour[] = date('d');
  $retour[] = date('m');
  $retour[] = date('Y');

  // Renvoi du tableau
  return $retour;
}
?>
```

L'exemple ci-dessous montre l'utilisation des deux fonctions précédentes. On utilise la décomposition du tableau en retour grâce à l'opérateur `list`, mais on pourrait également récupérer une seule variable de type tableau, et la traiter ensuite :

Exemple 2.7 *exemples/QuelJour.php* : *Appel des fonctions précédentes*

```
<?php
// Exemple d'appel à une fonction renvoyant plusieurs
// valeurs: passage par référence et passage par tableau

require_once('RenvoiTableau.php');
require_once('References.php');

// On veut obtenir le jour, le mois, l'an.
$an = $mois = $jour = "";

// Passage des valeurs par référence
aujourdhui_ref ($jour, $mois, $an);
echo "Nous sommes le $jour/$mois/$an<br/>";

// Appel, et récupération des valeurs du tableau
```

```
list($jour, $mois, $an) = aujourdhui_tab();

echo "Confirmation: nous sommes le $jour/$mois/$an<br/>";
?>
```

Une caractéristique de cette syntaxe est que l'on *ne sait pas*, en regardant ce code, que la fonction `aujourdhui_ref()` modifie les valeurs de ses paramètres et a donc un impact invisible sur le script appelant. Si on commence à utiliser le passage par référence pour certaines fonctions, on se retrouve donc dans un monde incertain où certaines variables sont modifiées après un appel à une fonction sans que l'on sache pourquoi. Du point de vue de la compréhension du code, le passage des paramètres par valeur est donc préférable.

Ce qui nous amène au second argument en faveur du passage par référence : le passage par valeur entraîne des copies potentiellement pénalisantes. Cet argument est à prendre en considération si on pense que la fonction en cause est appelée très fréquemment et manipule des données volumineuses, mais on doit être conscient que le recours aux références est plus délicat à manipuler et rend le code moins sûr.

Dans le cadre de ce livre, où la lisibilité des exemples et du code est un critère primordial, aucune fonction n'utilise de passage par référence (et encore moins de variable globale). Cela montre, incidemment, qu'il est tout à fait possible de se passer totalement de ces mécanismes. Dans la plupart des cas on y gagne un code sain, sans impacter les performances. Je reviendrai à quelques occasions sur ce choix pour discuter d'une autre stratégie de développement consistant à recourir au passage par références. Comme indiqué ci-dessus, les objets sont une exception en PHP : ils sont toujours passés par référence.

2.2 TRAITEMENT DES DONNÉES TRANSMISES PAR HTTP

Pour étudier de manière concrète les problèmes soulevés par l'échange de données via HTTP (et leurs solutions), nous allons étudier une application très simplifiée d'envoi de courrier électronique (terme que nous simplifierons en e-mail) dont la figure 2.1 donne le schéma. Il s'agit d'un script unique, *Mail.php*, qui fonctionne en deux modes :

1. Si aucune donnée ne lui est soumise, le script affiche un formulaire de saisie d'un e-mail. L'utilisateur peut alors entrer les données du formulaire et les soumettre. Elles sont transmises, au même script, via HTTP.
2. Si des données sont soumises (cas où on est donc passé par le mode précédent), le script récupère les données et doit :
   - envoyer l'e-mail,
   - stocker l'e-mail dans la base de données,
   - l'afficher en HTML pour confirmer la prise en charge de l'envoi.

Ce qui nous intéresse ici, c'est le traitement des données transférées dans trois contextes différents : envoi sous forme de texte pur, insertion dans MySQL et

affichage avec HTML. Chaque traitement est implanté par une fonction détaillée dans ce qui suit. Voici le script général, *Mail.php*, qui appelle ces différentes fonctions.

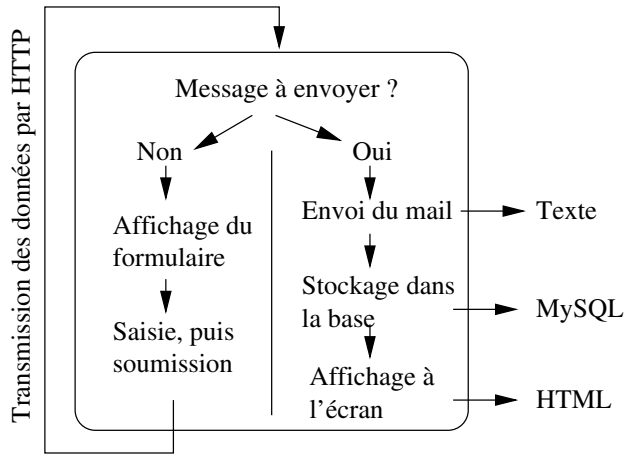


Figure 2.1 — Le schéma de l'application d'envoi d'un e-mail

Exemple 2.8 *exemples/Mail.php* : *Script de gestion d'un e-mail*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<head>
<title>Envoi d'un e-mail</title>
<link rel='stylesheet' href="films.css" type="text/css" />
</head>
<body>

<h1>Envoi de mail</h1>

<?php
// Inclusion des fichiers contenant les déclarations de fonctions

require_once("Normalisation.php");
require_once("ControleMail.php");
require_once("StockeMail.php");
require_once("AfficheMail.php");
require_once("EnvoiMail.php");

// Normalisation des entrées HTTP
Normalisation();

// Si la variable $envoyer existe, des données ont été saisies
// dans le formulaire
```

```
if (isSet($_POST['envoyer'])) {
  // Contrôle des données en entrée
  if (!ControleMail($_POST)) {
    // Un problème quelque part? Il faut réagir
    echo "<p>Quelque chose ne va pas...</p>";
    exit;
  }

  // On a passé le test: stockage dans la base
  StockeMail($_POST);

  // On affiche le texte de l'e-mail
  AfficheMail ($_POST);

  // Envoi de l'e-mail
  EnvoiMail($_POST);

}
else {
  // On affiche simplement le formulaire
  require ("FormMail.html");
}
?>
</body>
</html>
```

Le premier mode de l'application, avec le formulaire de saisie, est présenté figure 2.2.

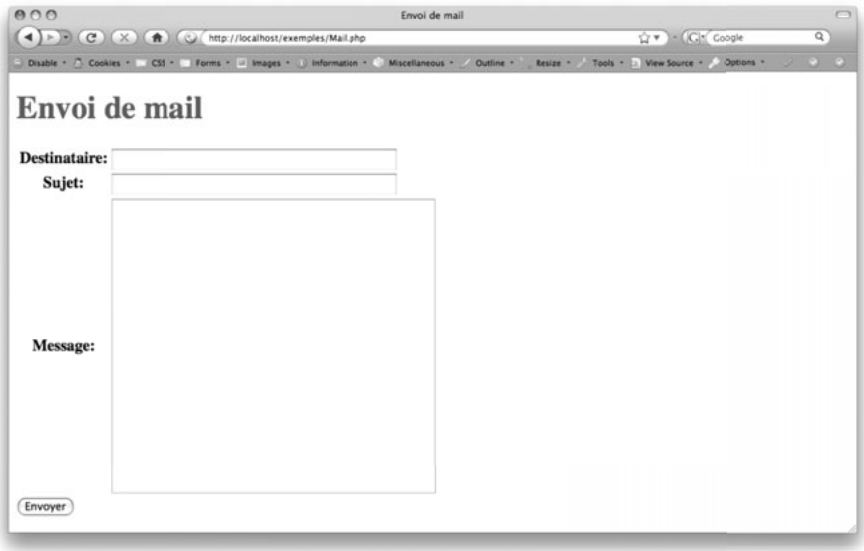


Figure 2.2 — Formulaire d'envoi d'un e-mail

2.2.1 Échappement et codage des données HTTP

Quand l'utilisateur soumet un formulaire, le navigateur code les données saisies dans les champs pour les inclure dans un message HTTP. Rappelons que :

- en mode `get` les paramètres sont placés dans l'URL appelée ;
- en mode `post` les paramètres sont transmis dans le corps du message HTTP.

Nous allons utiliser le mode `post` car la méthode `get` a l'inconvénient de créer des URL très longues pour y stocker les paramètres. Une partie de ces derniers peut d'ailleurs être perdue si la taille limite (256 caractères selon le protocole HTTP) est dépassée. Un mode de passage des paramètres imposant le mode `get` est celui où l'on place directement les paramètres dans une ancre du document HTML. Supposons par exemple qu'on place quelque part dans le document HTML une ancre dans laquelle on passe le sujet de l'e-mail :

```
echo "<a href='Mail.php?sujet=$sujet'>Message</a>";
```

Il faut être attentif dans ce cas à bien coder l'URL selon les règles HTTP. Si le sujet est par exemple `Wallace & Gromit ou l'ours ?`, l'esperluette « & » et le point d'interrogation « ? », placés littéralement dans l'URL, rendront le message HTTP incompréhensible pour le script. On obtiendra en effet :

```
echo "<a href='Mail.php?sujet=Wallace & Gromit ou l'ours ?>Message</a>";
```

Le codage s'obtient en appliquant la fonction `urlEncode()` aux chaînes placées dans les ancres. Voici donc la bonne version :

```
$sujet = "Wallace & Gromit ou l'ours ?";
$sujetPourURL = urlEncode($sujet);
echo "<a href='Mail.php?sujet=$sujetPourURL'>Message</a>";
```

L'URL sera alors codée comme suit :

```
Mail.php?sujet=Wallace+%5C%26+Gromit+ou+l%27ours+%3F
```

Revenons au cas où l'on utilise un formulaire, ce qui garantit que le navigateur effectuera le codage pour nous. Après soumission, le message est transmis au script indiqué dans l'attribut `action` du formulaire. Dans notre cas, le script est « réentrant » car il reçoit lui-même les données soumises par le formulaire qu'il a affiché. Il suffit d'être capable de déterminer, à l'entrée du script, si le formulaire vient d'être soumis ou non. On utilise un champ caché, de nom `envoyer`, qui provoquera donc l'instanciation d'une variable PHP après soumission.

Exemple 2.9 *exemples/FormMail.html* : *Le formulaire de saisie des e-mails*

```
<!-- Formulaire basique pour l'envoi d'un e-mail -->

<form action='Mail.php' method='post'>

<!-- Champ caché pour indiquer que le formulaire a été soumis -->
```

```
<input type='hidden' name='envoyer' value='1'/>

<table>
 <tr><th>Destinataire:</th>
     <td><input type='text' size='40' name='destinataire'/></td>
 </tr>
 <tr><th>Sujet:</th>
     <td><input type='text' size='40' name='sujet'/></td>
 </tr>
 <tr><th>Message:</th>
     <td><textarea rows='20' cols='40' name='message'></textarea>
     </td>
 </tr>
</table>
<input type='submit' value='Envoyer'/>
</form>
```

À l'entrée du script, le processeur PHP décrypte les données provenant du formulaire et transférées dans le message HTTP, puis les place dans les tableaux `$_POST` ou `$_GET` selon le mode de transmission choisi. On peut également utiliser systématiquement le tableau `$_REQUEST` qui fusionne les deux précédents (plus le tableau `$_COOKIES`).

À ce stade, PHP peut effectuer ou non une transformation consistant à préfixer toutes les apostrophes simples ou doubles par la barre oblique inverse « \ ». Ce comportement est déterminé par l'option de configuration `magic_quotes_gpc`, et motivé par l'insertion fréquente des données provenant de formulaires dans des requêtes SQL. Un des points épineux dans la manipulation de chaînes de caractères insérées ou lues dans une base MySQL est en effet la présence d'apostrophes. Prenons l'exemple suivant :

```
INSERT INTO FilmSimple (titre, annee, nom_realisateur,
                        prenom_realisateur, annee_naissance)
VALUES    ('L'ours', 1988, 'Annaud', 'Jean-Jacques', 1943)
```

MySQL distingue les valeurs grâce aux apostrophes simples « ' ». Si une valeur contient elle-même une apostrophe, comme « L'ours » dans l'exemple ci-dessus, MySQL est perdu et produit un message d'erreur. La bonne syntaxe est :

```
INSERT INTO FilmSimple (titre, annee, nom_realisateur,
                        prenom_realisateur, annee_naissance)
VALUES    ('L\'ours', 1988, 'Annaud', 'Jean-Jacques', 1943)
```

La présence d'un caractère « \ » devant l'apostrophe (on parle « d'échappement ») permet à MySQL d'interpréter correctement cette dernière comme faisant partie de la chaîne. Quand l'option `magic_quotes_gpc` vaut `On`, un titre comme `L'ours` sera automatiquement représenté par la valeur `L\'ours` dans le script recevant les données. On pourra donc l'insérer tel quel dans une requête SQL.

Cependant, comme le montre l'application que nous sommes en train de créer, une chaîne de caractères peut être utilisée dans bien d'autres contextes que SQL, et

« l'échappement » des apostrophes par des barres obliques devient inutile et gênant. Il faut alors se poser sans cesse la question de la provenance de la variable, de la configuration courante de PHP, et de la nécessité ou non d'utiliser l'échappement. C'est encore plus ennuyeux quand on écrit des fonctions puisqu'il faut déterminer si les paramètres peuvent ou non provenir d'une transmission HTTP, et si oui penser à uniformiser, dans les appels à la fonction, la règle d'échappement à utiliser.

Depuis la parution de PHP 5, les concepteurs et distributeurs du langage semblent renoncer à cet échappement automatique. Le problème est de risquer de se trouver dans une situation où certain serveurs pratiquent l'échappement et d'autres non.

Le seul moyen pour régler le problème une fois pour toutes et de *normaliser* systématiquement les données HTTP. La politique adoptée dans ce livre (vous êtes libre d'en inventer une autre bien entendu) consiste à tester, à l'entrée de tout script, si le mode d'échappement automatique est activé. Si oui, on supprime cet échappement, pour toutes les chaînes transmises, avec la fonction `stripSlashes()`. On pourra alors considérer par la suite que les données HTTP sont représentées normalement, comme n'importe quelle autre chaîne de caractères manipulée dans le script. Voici la fonction qui effectue cette opération sur chacun des tableaux contenant d'une part des données transmises en mode `get` ou `post`, d'autre part des *cookies*.

Exemple 2.10 *exemples/Normalisation.php* : *Traitement des tableaux pour supprimer l'échappement automatique*

```
<?php
// Application de la suppression des échappements, si nécessaire,
// dans tous les tableaux contenant des données HTTP

require_once("NormalisationHTTP.php");

function Normalisation()
{
  // Si l'on est en échappement automatique, on rectifie...
  if (get_magic_quotes_gpc()) {
    $_POST = NormalisationHTTP($_POST);
    $_GET = NormalisationHTTP($_GET);
    $_REQUEST = NormalisationHTTP($_REQUEST);
    $_COOKIE = NormalisationHTTP($_COOKIE);
  }
}
?>
```

La fonction `get_magic_quotes_gpc()` indique si l'échappement automatique est activé. On parcourt alors les tableaux concernés et traite chaque valeur[1] avec `stripSlashes()` qui supprime les « \ ». Dans le parcours lui-même, il faut prendre

1. On ne traite pas la *clé* de chaque élément, en considérant qu'une clé ne devrait pas contenir d'apostrophes.

en compte le fait qu'un élément du tableau peut constituer lui-même un tableau imbriqué (cas par exemple d'un formulaire permettant de saisir plusieurs valeurs pour un champ de même nom, voir page 46). Une manière simple et naturelle de parcourir les tableaux imbriqués sans se soucier du nombre de niveaux est d'appeler récursivement la fonction de normalisation `NormalisationHTTP()`, donnée ci-dessous.

Exemple 2.11 *exemples/NormalisationHTTP.php* : *Parcours récursif des tableaux pour appliquer* `stripSlashes()`.

```
<?php
// Cette fonction supprime tout échappement automatique
// des données HTTP dans un tableau de dimension quelconque

function NormalisationHTTP($tableau)
{
  // Parcours du tableau
  foreach ($tableau as $cle => $valeur)
    {
      if (!is_array($valeur)) // c'est un élément: on agit
        $tableau[$cle] = stripSlashes($valeur);
      else  // c'est un tableau : on appelle récursivement
        $tableau[$cle] = NormalisationHTTP($valeur);
    }
  return $tableau;
}
?>
```

La construction `foreach` utilisée ici est très pratique pour parcourir un tableau en récupérant à la fois l'indice et la valeur de chaque entrée. On peut noter que cette fonction prend en entrée un tableau et produit en sortie une copie dans laquelle les échappements éventuels ont été supprimés. Il est possible, si l'on considère que ces copies sont pénalisantes, de traiter les paramètres par référence en les préfixant par « & ».

2.2.2 Contrôle des données HTTP

La seconde tâche à effectuer en recevant des données d'un formulaire est le contrôle des données reçues. Cela signifie, au minimum,

1. le test de l'existence des données attendues,
2. un *filtrage* sur ces données, afin de supprimer des caractères parasites qui pourraient infecter l'application;
3. et enfin le contrôle de quelques caractéristiques minimales sur les valeurs.

On n'insistera jamais assez sur le fait qu'un script PHP est un programme que le monde entier peut appeler en lui passant n'importe quoi. Bien entendu la majorité des utilisateurs du Web a bien autre chose à faire que d'essayer de casser votre application, mais il suffit d'un malveillant pour créer des problèmes, et de plus ces

attaques sont malheureusement automatisables. Un jour ou l'autre vous serez amenés à vous poser la question de la robustesse de vos scripts. Le filtrage des données en entrée, en particulier, est très important pour les sécuriser.

La fonction ci-dessous est une version minimale des contrôles à effectuer. Elle repose pour les contrôles sur les fonctions `isSet()` et `empty()` qui testent respectivement l'existence d'une variable et la présence d'une valeur (chaîne non vide). Pour le filtrage la fonction utilise `htmlSpecialChars()` qui remplace les caractères marquant une balise (soit « < », « > » et « & ») par un appel d'entité (soit, respectivement, `<`, `>` et `&`). On peut également envisager de supprimer totalement les balises avec la fonction `strip_tags()`. L'injection de balises HTML dans les champs de formulaires est une technique classique d'attaque d'un site web.

La fonction prend en entrée un tableau contenant les données, renvoie `true` si elles sont validées, et `false` sinon. Remarquer que le tableau `$mail` est passé par référence pour permettre sa modification suite au filtrage. On pourra utiliser cette fonction en lui passant le tableau `$_POST` pour valider la saisie du formulaire précédent, ainsi que tout autre tableau dont on voudrait contrôler le contenu selon les mêmes règles. Il est toujours préférable de concevoir des fonctions les plus indépendantes possibles d'un contexte d'utilisation particulier.

Exemple 2.12 *exemples/ControleMail.php* : *Ébauche de contrôle des données*

```
<?php
 // Fonction contrôlant l'entrée de l'application e-mail.

 function ControleMail (&$mail)
 {
   // Le tableau en paramètre doit contenir les entrées:
   // destinataire, sujet et message. Vérification.
   if (!isSet($mail['destinataire']))
     {echo "Pas de destinataire!"; return false;}
   else $mail['destinataire'] = htmlSpecialChars($mail
       ['destinataire']);

   if (!isSet($mail['sujet']))
     {echo "Pas de sujet!"; return false;}
   else $mail['sujet'] = htmlSpecialChars($mail
   ['sujet']);

   if (!isSet($mail['message']))
     {echo "Pas de message!"; return false;}
   else $mail['message'] = htmlSpecialChars($mail['message']);

   // On vérifie que les données ne sont pas vides
   if (empty($mail['destinataire']))
     {echo "Destinataire vide!"; return false;}
   if (empty($mail['sujet']))
     {echo "Sujet vide!"; return false;}
   if (empty($mail['message']))
     {echo "Message vide!"; return false;}
```

```
    // Maintenant on peut/doit également faire des contrôles
    // sur les valeurs attendues: destinataire, sujet, message.
    // Voir les exercices pour des suggestions.

    return true;
  }
?>
```

On pourrait envisager beaucoup d'autres contrôles à effectuer, certains étant décrits dans le document d'exercices disponible sur le site. Les contrôles s'appuient fréquemment sur la vérification du format des données (comme, typiquement, l'adresse électronique) et nécessitent le recours aux expressions régulières qui seront présentées page 87.

Dans toute la suite de ce livre, j'omets le plus souvent de surcharger le code par des contrôles répétitifs et nuisant à la clarté du code. Le filtrage et le contrôle des données en entrée font partie des impératifs de la réalisation d'un site sensible : reportez-vous au site `php.net` pour des recommandations à jour sur la sécurité des applications PHP.

2.2.3 Comment insérer dans la base de données : insertion dans MySQL

Voyons maintenant comment effectuer des insertions dans la base à partir des données reçues. Il faut tout d'abord créer une table, ce qui se fait avec le script SQL suivant :

Exemple 2.13 *exemples/Mail.sql* : *Création de la table stockant les e-mails*

```
#
# Création d'une table pour stocker des e-mails
#

CREATE TABLE Mail (id_mail INT AUTO_INCREMENT NOT NULL,
                   destinataire VARCHAR(40) NOT NULL,
                   sujet VARCHAR(40) NOT NULL,
                    message TEXT NOT NULL,
                    date_envoi DATETIME,
                    PRIMARY KEY (id_mail));
```

Petite nouveauté : on trouve dans la table *Mail* une option `AUTO_INCREMENT`, spécifique à MySQL. Cette option permet d'incrémenter automatiquement l'attribut `id_mail` à chaque insertion. De plus, cet attribut doit être déclaré comme clé primaire, ce qui signifie qu'il ne peut pas prendre deux fois la même valeur parmi les lignes de la table. On peut insérer une ligne dans *Mail* sans indiquer de valeur pour `id_mail`, déterminée automatiquement par MySQL.

```
mysql> INSERT INTO Mail (destinataire, sujet, message, date_envoi)
    ->  VALUES ('rigaux@lri.fr', 'Essai', 'Test du mail', NOW());
Query OK, 1 row affected (0,00 sec)
```

La fonction `LAST_INSERT_ID()` permet de savoir quelle est la dernière valeur générée pour un champ `AUTO_INCREMENT`.

```
mysql> SELECT LAST_INSERT_ID() ;
+------------------+
| last_insert_id() |
+------------------+
|               36 |
+------------------+
1 row in set (0.06 sec)
mysql>
```

Enfin, on peut vérifier qu'il est impossible d'insérer deux fois un e-mail avec le même identifiant.

```
mysql> INSERT INTO Mail (id_mail, destinataire, sujet,
    ->                       message, date_envoi)
    ->  VALUES (36, 'rigaux@dauphine.fr', 'Essai', 'Test du mail', NOW());
ERROR 1062 (23000): Duplicate entry '36' for key 1
```

Nous reviendrons sur ces questions – essentielles – d'identification dans le chapitre 4.

La fonction ci-dessous prend un tableau en paramètre, traite ses entrées en échappant les apostrophes, et exécute enfin la requête.

Exemple 2.14 *exemples/StockeMail.php* : *Commande d'insertion dans MySQL*

```
<?php
  require_once ("UtilBD.php");

 // Fonction stockant un e-mail dans la base. Le tableau en
 // paramètre doit contenir les entrées destinataire, sujet
 // et message. NB: il faudrait vérifier les valeurs.

 function StockeMail ($mail)
 {
  // Connexion au serveur
  $connexion = Connexion (NOM, PASSE, BASE, SERVEUR);

  // On "échappe" les caractères gênants.
  $destinataire = mysql_real_escape_string($mail['destinataire']);
  $sujet = mysql_real_escape_string($mail['sujet']);
  $message = mysql_real_escape_string($mail['message']);

  // Création et exécution de la requête
```

```
  $requete = "INSERT INTO Mail(destinataire, sujet, message,
      date_envoi) "
    . "VALUES ('$destinataire', '$sujet', '$message', NOW())";

  ExecRequete ($requete, $connexion);
 }
?>
```

Suite à notre décision d'éliminer tout échappement automatique en entrée d'un script PHP, il faudra penser systématiquement à traiter avec la fonction `mysql_real_escape_string()` les chaînes à insérer dans MySQL [2].

2.2.4 Traitement de la réponse

Avant afficher, dans un document HTML, un texte saisi dans un formulaire, il faut se poser les questions suivantes :

- Quelle sera la mise en forme obtenue ? Rend-elle correctement la saisie de l'utilisateur ?
- Le texte peut-il contenir lui-même des balises HTML qui vont gêner l'affichage ?

Voici un exemple de texte que l'utilisateur pourrait saisir dans le formulaire, potentiellement source de problème :

```
Pour créer un formulaire, on utilise la balise <form> et une
suite de balises <input>. Voici un exemple ci-dessous:

<form action='monscript'>
  <input type=text name=='n1' size='10'/>
  <input type=text name=='n2' size='10'/>
  <input type='submit'/>
</form>
Bonne réception!

PR
```

Quand on transmet le texte saisi dans une fenêtre de formulaire au script et que ce dernier le renvoie au navigateur pour l'afficher, on obtient le résultat de la figure 2.3. Les balises apparaissent littéralement et ne sont pas interprétées par le navigateur. Pourquoi ? Parce que nous les avons traitées avec `htmlSpecialChars()` et que le texte `<input>` a été remplacé par `<input>`. Faites l'essai, et retirez le filtrage par `htmlspecialChars()` pour constater les dégâts à l'affichage.

2. Il est d'usage d'appeler `addSlashes()` qui suffira dans la très grande majorité des cas, mais `mysql_real_escape_string()` est un peu plus complète et adaptée à MySQL, pour la prise en compte des jeux de caractères par exemple.

Par ailleurs, pour conserver les sauts de ligne en HTML, il faut insérer explicitement des balises <**br/**>. PHP fournit une fonction, `nl2br()`, qui permet de convertir les caractères de sauts de ligne en balises <**br/**>, préservant ainsi l'affichage.

Figure 2.3 — Affichage du texte d'un e-mail comprenant des balises

La chaîne de caractères obtenue après ces traitements prophylactiques est donnée ci-dessous, ce qui permet d'afficher le résultat donné figure 2.3.

```
<h1>Envoi de mail</h1>

<b>On a envoyé le message suivant: </b>
<p>Pour créer un formulaire, on utilise<br />
la balise &lt;form&gt; et une suite de balises<br />
&lt;input&gt;. Voici un exemple ci-dessous:<br />
<br />
&lt;form action='monscript'&gt;<br />
  &lt;input type=text name=='n1' size='10'&gt;<br />
  &lt;input type=text name=='n2' size='10'&gt;<br />
  &lt;input type='submit'&gt;<br />
&lt;/form&gt;<br />
Bonne réception!<br />
<br />
PR
```

La présence de lettres accentuées dans un document HTML ne pose pas de problème à un navigateur employant le jeu de caractères standard occidental ou l'UTF-8. Rappelons que ce réglage est spécifié au début du document avec l'option suivante pour Latin1 :

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Une autre possibilité est de remplacer toutes les lettres accentuées (et de manière générale tous les caractères spéciaux) par un appel à l'entité correspondante (par exemple « é » devient « `é` »). On obtient ce remplacement avec la fonction `htmlEntities()`.

2.2.5 Comment obtenir du texte « pur » : envoi de l'e-mail

Finalement, il reste à envoyer l'e-mail, grâce à la fonction `mail()` de PHP[3]. Les questions à se poser sont ici réciproques de celles étudiées ci-dessus pour le passage d'une représentation en texte brut à une représentation HTML. Si le texte à envoyer contient des mises en forme HTML, on peut les supprimer avec la fonction `strip_tags()`, comme le montre la fonction ci-dessous.

Exemple 2.15 *exemples/EnvoiMail.php* : *Fonction d'envoi d'un e-mail*

```
<?php
 // Fonction envoyant un e-mail. On suppose
 // que les contrôles ont été effectués avant l'appel à la
 // fonction

 function EnvoiMail ($mail)
 {
  // Extraction des paramètres
  $destinataire = $mail['destinataire'];
  $sujet = $mail['sujet'];

  // On retire toutes les balises HTML du message
  $message = strip_tags($mail['message']);

  // On va indiquer l'expéditeur, et placer rigaux@dauphine.fr en
  // copie
  $entete = "From: mysqlphp@dunod.fr\r\n";
  $entete .= "Cc: rigaux@dauphine.fr\r\n";

  // Appel à la fonction PHP standard
  mail ($destinataire, $sujet, $message, $entete);
 }
?>
```

L'étude de fonctionnalités plus avancées d'envoi d'e-mails (avec fichiers en attachement par exemple) dépasse le cadre de ce livre. Comme d'habitude je vous renvoie à `php.net`, ou à des fonctionnalités prêtes à l'emploi comme *phpMailer* (voir le site *developpez.com*).

3. Cette fonction nécessite l'accès à un serveur SMTP, et peut être désactivée chez votre fournisseur d'accès pour éviter l'envoi de *spams* (ou pourriels).

2.2.6 En résumé : traitement des requêtes et des réponses

Le cas d'école qui précède montre les principes règles à appliquer aux chaînes de caractères transmises par HTTP, et aux réponses transmises au client. Le tableau 2.1 rappelle pour sa part la liste des fonctions essentielles au traitement des données HTTP.

1. s'assurer, à l'entrée du script, que les chaînes suivent toujours la même règle d'échappement ; étant donné que la configuration peut varier, le seul moyen sûr est d'effectuer un pré-traitement à l'entrée dans le script (fonction `NormalisationHTTP()`) ;
2. effectuer tous les contrôles nécessaires sur la présence et les valeurs des données transmises ;
3. filter les données en entrées, en supprimant notamment les balises avec `strip_tags()` ou en les neutralisant avec `htmlSpecialChars()` ;
4. utiliser un échappement (avec « \ ») avant d'insérer dans la base MySQL ;
5. appliquer un échappement aux caractères réservés HTML (« < », « > », « & »), en les transformant en appels d'entités avant de transmettre à un navigateur ;
6. supprimer les balises HTML avant un affichage en mode texte (ou envoi d'un e-mail, ou toute autre situation comparable).

Tableau 2.1 — Fonctions utilisées dans cette section

Fonction	Description
get_magic_quotes_gpc	Renvoie « vrai » si les guillemets et apostrophes sont automatiquement « échappées » dans les chaînes transmises par HTTP, faux sinon.
isSet(*nom variable*)	Renvoie vrai si la variable est définie, faux sinon.
empty(*nom variable*)	Renvoie vrai si la variable n'est pas la chaîne vide, faux sinon.
htmlEntities(*chaîne*)	Renvoie une chaîne où les caractères spéciaux présents dans *chaîne* sont remplacés par des entités.
htmlSpecialChars(*chaîne*)	Renvoie une chaîne où les caractères « < », « > », « & », « ' » et « " » présents dans *chaîne* sont remplacés par des entités.
strip_tags(*chaîne*, [*balises*])	Renvoie une chaîne où les balises HTML présentes dans *chaîne* sont supprimées (à l'exception de celles données dans le second argument, optionnel).
addSlashes(*chaîne*)	Renvoie une chaîne où les guillemets et apostrophes sont préfixées par « \ », notamment en vue de l'insertion dans une base de données.
mysql_real_escape_string(*chaîne*)	Idem que la précédente, mais adaptée à MySQL pour le traitement de données binaires ou de lettres accentuées.
stripSlashes(*chaîne*)	Fonction inverse de la précédente : renvoie une chaîne où les barres « \ » sont supprimées devant les guillemets et apostrophes.
nb2br(*chaîne*)	Renvoie une chaîne où les caractères ASCII de fin de ligne (\n) sont remplacés par la balise .
urlEncode(*chaîne*)	Renvoie une chaîne codée pour pouvoir être insérée dans une URL.

Est-il nécessaire de préciser qu'il faut rester extrêmement prudent avec les données transmises par HTTP ? Il est par exemple très délicat de proposer un formulaire pour saisir des commandes à effectuer, dans MySQL ou dans le système d'exploitation. Encore une fois, reportez-vous au site `php.net` pour sa documentation sur la sécurité des applications PHP, et de nombreuses recommandations à ce sujet.

2.3 MISE À JOUR D'UNE BASE PAR FORMULAIRE

L'interface de mise à jour de la table *FilmComplet* donnée à la fin du chapitre 1 (voir les exemples pages 47 et 49) est assez rudimentaire et ferait rapidement hurler n'importe quel utilisateur. Nous allons développer un système plus convivial pour insérer, mettre à jour ou détruire les lignes d'une table, en prenant comme cible la table *FilmSimple*, qui est un peu plus facile à manipuler (voir le schéma page 28).

2.3.1 Script d'insertion et de mise à jour

Le script principal, *FilmSimple.php*, affiche une page dont le contenu varie en fonction du mode choisi. Voici les modes possibles :

1. En mode *par défaut*, on affiche la liste des films en leur associant une ancre permettant d'accéder au formulaire de modification. Une ancre placée sous le tableau permet d'accéder au formulaire d'insertion.
2. En mode *modification* d'un film, on affiche un formulaire présentant les champs de saisie. Chaque champ vaut par défaut la valeur couramment stockée dans la base pour ce film. Seule exception : on ne peut pas modifier le titre puisqu'on suppose ici que c'est le moyen *d'identifier* (et donc de retrouver) le film modifié.
3. Enfin, en mode *insertion*, on présente un formulaire de saisie, sans valeur par défaut.

Pour commencer nous allons définir avec `define()` des constantes définissant les différents modes. Les constantes permettent de manipuler des symboles, plus faciles à utiliser et plus clairs que des valeurs.

```
// Les constantes pour le mode
define ("MODE_DEFAUT", "defaut");
define ("MODE_INSERTION", "insertion");
define ("MODE_MAJ", "maj");
```

Ensuite, afin de ne pas se lancer dans un script d'une taille démesurée, on découpe le travail en plusieurs parties, correspondant chacune à une fonctionnalité précise, puis on réalise chaque partie par une fonction.

La première fonction affiche le formulaire. On pourrait prévoir une fonction pour un formulaire en mise à jour et une autre pour un formulaire en insertion, mais la plus grande part du code serait commun, ce qui entraîne une double modification chaque fois que le site évolue (par exemple lors de l'ajout d'un champ dans la table).

Il est beaucoup plus astucieux de programmer une seule fonction qui affiche un contenu légèrement différent en fonction du type de mise à jour souhaité (insertion ou modification). Voici le code de cette fonction. Le style de programmation adopté ici est du HTML dans lequel on insère ponctuellement des instructions PHP. Ce style trouve très rapidement ses limites en terme de lisibilité, comme vous pourrez vous en convaincre en essayant de décrypter le contenu. Rassurez-vous, c'est la dernière fois que j'utilise cette gestion obscure des accolades !

Exemple 2.16 *exemples/FormFilmSimple.php* : *Le formulaire avec valeurs par défaut, et modes insertion ou mise à jour*

```
<?php
 // Formulaire de saisie, avec valeurs par défaut

 function FormFilmSimple ($mode, $val_defaut)
 {
  ?>
   <!-- On est en HTML          -->
   <form action='FilmSimple.php' method='post'>
   <input type='hidden' name="action" value="FormFilmSimple"/>
   <input type='hidden' name="mode" value="<?php echo $mode ?>"/>
   <table>

   <?php if ($mode == MODE_INSERTION) { ?>
      <tr><td>Titre : </td><td><input type='text' size='40' name=
          'titre' value="<?php echo $val_defaut['titre']?>"/>
                        </td></tr>
   <?php } else { ?>
     <tr><td>Mise à jour de </td><td><?php echo $val_defaut
         ['titre']?>
     <input type='hidden'
            name='titre'
                  value='<?php echo $val_defaut['titre']?>'/>
                         </td></tr>
    <?php } ?>

   <tr><td>Année : </td>
       <td><input type='text' size='4' maxlength='4'
          name="annee" value="<?php
                                echo $val_defaut['annee']?>"/>
                                </td></tr>

   <tr><td>Réalisateur (prénom – nom) : </td>
       <td><input type='text' size='20' name="prenom_realisateur"
             value="<?php
                      echo $val_defaut['prenom_realisateur']?>"/>
                         <br/>
            <input type=text size='20' name="nom_realisateur"
             value="<?php echo $val_defaut['nom_realisateur']?>"/>
                    </td></tr>
```

```
    <tr><td>Année de naissance :
        <td><input type='text' size='4' maxlength='4'
              name='annee_naissance'
              value="<?php echo $val_defaut['annee_naissance']?>"/>
                    </td></tr>

    <tr><td colspan='2'><input type='submit' value='Exécuter'/>
        </td></tr>
   </table>
   </form>
<?php
}
?>
```

La fonction `FormFilmSimple()` prend en paramètres le mode (insertion ou modification) et un tableau contenant les valeurs par défaut à placer dans les champs. Le mode est systématiquement placé dans un champ caché pour être transmis au script traitant les données du formulaire, et saura ainsi dans quel contexte elles ont été saisies :

```
<input type='hidden'name="mode"value="<?php echo $mode ?>"/>
```

Les modes d'insertion et de modification correspondent à deux affichages différents du formulaire. En insertion le champ « titre » est saisissable et sans valeur par défaut. En mise à jour, il est affiché sans pouvoir être saisi, et il est de plus dans un champ caché avec sa valeur courante qui est transmise au script de traitement des données.

```
<?php if ($mode == MODE_INSERTION) { ?>
    <tr><td>Titre : </td><td><input type='text' size='40' name='
        titre'
                          value="<?= $val_defaut['titre']?>"/></td
                             ></tr>
<?php } else { ?>
   <tr><td>Mise à jour de </td><td><?= $val_defaut['titre']?></td>
   <td><input type='hidden'
        name='titre' value='<?= $val_defaut['titre']?>'/></td></tr
            >
 <?php } ?>
```

Le tableau des valeurs par défaut passé en paramètre à la fonction, `$val_defaut`, doit contenir un élément par champ, le nom de l'élément étant le nom du champ, et sa valeur la valeur par défaut du champ. Ce tableau associatif peut s'obtenir par un appel à `mysql_fetch_assoc()` si le film vient de la base et doit être modifié. Il peut également s'agir du tableau `$_POST` ou `$_GET` après saisie du formulaire, pour réafficher les données prises en compte. La figure 2.4 montre le formulaire en modification.

Notez qu'on place un autre champ caché, `action`, dans le formulaire. La transmission de cette variable `action` au script *FilmSimple.php* indique que des valeurs ont

été saisies dans le formulaire, et qu'elles doivent déclencher une insertion ou une mise à jour dans la base.

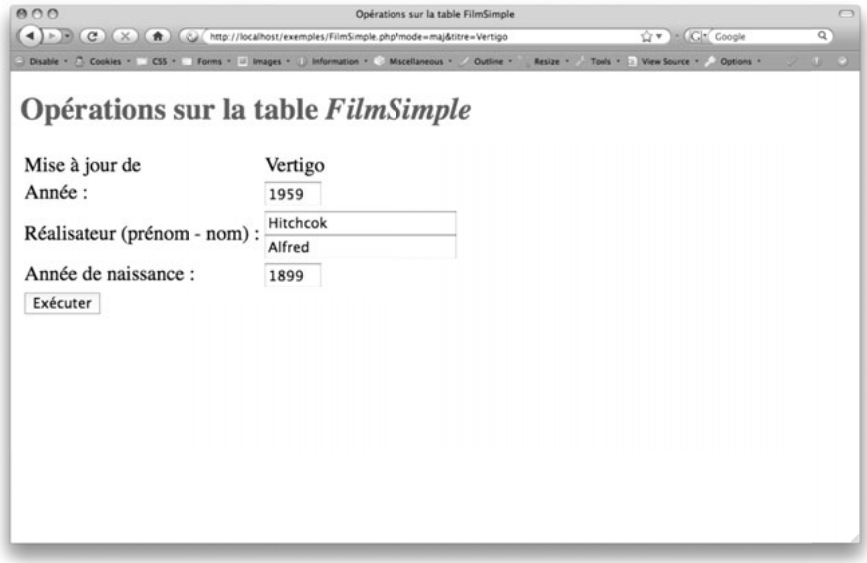


Figure 2.4 — Formulaire en modification du film *Vertigo*

La seconde fonction effectue les requêtes de mise à jour. Elle prend en entrée le mode (insertion ou modification), un tableau associatif qui, comme `$val_defaut`, contient les valeurs d'une ligne de la table *FilmSimple*, enfin l'identifiant de connexion à la base.

Exemple 2.17 *exemples/MAJFilmSimple.php* : *Fonction de mise à jour de la table*

```
<?php
// Fonction de mise à jour ou insertion de la table FilmSimple

function MAJFilmSimple ($mode, $film, $connexion)
{
  // Préparation des variables, en traitant par addSlashes
  // les chaînes de caractères
  $titre = addSlashes($film['titre']);
  $annee = $film['annee'];
  $prenom_realisateur = addSlashes($film['prenom_realisateur']);
  $nom_realisateur = addSlashes($film['nom_realisateur']);
  $annee_naissance = $film['annee_naissance'];

  if ($mode == MODE_INSERTION)
    $requete = "INSERT INTO FilmSimple (titre, annee, "
      . "prenom_realisateur, nom_realisateur, annee_naissance) "
      . " VALUES ('$titre', '$annee', '$prenom_realisateur', "
      . "'$nom_realisateur', '$annee_naissance')";
  else
```

```
    $requete = "UPDATE FilmSimple SET annee='$annee', "
      . "prenom_realisateur='$prenom_realisateur', "
      . "nom_realisateur='$nom_realisateur', "
      . "annee_naissance='$annee_naissance' "
      . "WHERE titre = '$titre' ";

  // Exécution de l'ordre SQL

  ExecRequete ($requete, $connexion);
}
?>
```

Enfin, la troisième fonction affiche un tableau des films présents dans la table, en associant à chaque film une ancre pour la modification.

Exemple 2.18 *exemples/TableauFilms.php* : *Tableau affichant la liste des films*

```
<?php
// Affichage du tableau des films

function TableauFilms ($connexion)
{
  $resultat = ExecRequete("SELECT * FROM FilmSimple", $connexion);

  echo "<table border='4' cellspacing='2' cellpadding='2'>"
  . "<caption align='bottom'>Table <i>FilmSimple</i></caption> "
  . "<tr><th>Titre</th><th>Année</th><th>Réalisateur</th>"
  . "<th>Année naissance</th><th>Action</th></tr>\n";

  while ($film = ObjetSuivant ($resultat)) {
    // On code le titre pour le placer dans l'URL
    $titreURL = urlEncode ($film->titre);
    echo "<tr><td>$film->titre</td><td>$film->annee</td> "
    . "<td>$film->prenom_realisateur $film->nom_realisateur</td> "
    . "<td>$film->annee_naissance</td> "
    . "<td><a href='FilmSimple.php?mode=" . MODE_MAJ
    . "&amp;titre=$titreURL'>Modifier ce film</a></td></tr>\n";
  }
  echo "</table>\n";
}
?>
```

Le tableau est standard, la seule particularité se limite à une l'ancre qui contient deux arguments, le mode et le titre du film à modifier :

```
<a href='FilmSimple.php?mode=maj&amp;titre=$titreURL'>
```

Rappelons qu'il faut prendre garde à ne pas placer n'importe quelle chaîne de caractères dans une URL (voir page 67) : des caractères blancs ou accentués seraient sans doute mal tolérés et ne donneraient pas les résultats escomptés. On ne doit

pas non plus placer directement un caractère « `&` » dans un document (X)HTML. On lui préférera la référence à l'entité « `&` ». Quand l'URL est produite dynamiquement, on doit lui appliquer la fonction PHP `urlEncode()` afin d'éviter ces problèmes.

Le script principal

Pour finir, voici le script principal, affichant la page de la figure 2.5. Toutes les instructions `require_once` pour inclure les différents fichiers de l'application, ainsi que les déclarations de constantes, ont été placées dans un seul fichier, *UtilFilmSimple.php*, inclus lui-même dans le script. Cela comprend notamment le fichier contenant la fonction `NormalisationHTTP()` (voir page 70) pour normaliser les entrées HTTP, que nous utiliserons maintenant systématiquement.

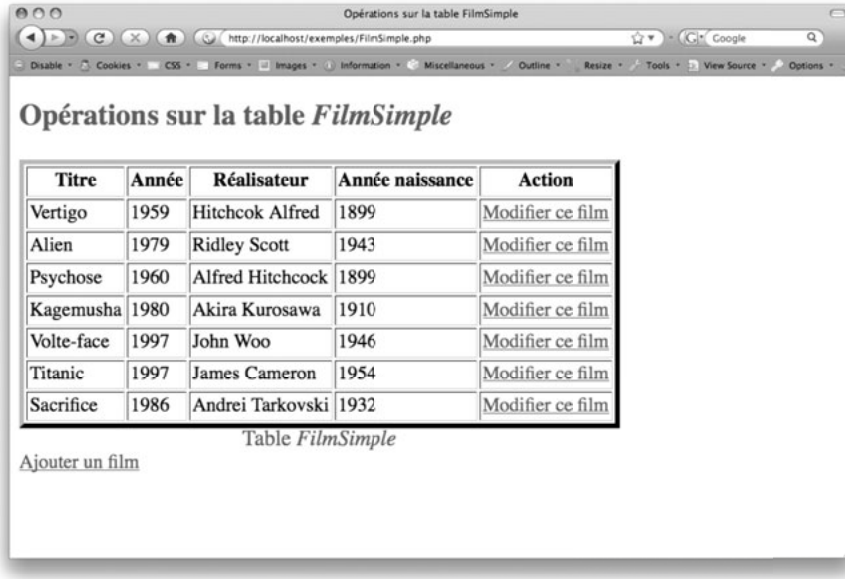

Opérations sur la table FilmSimple
http://localhost/exemples/FilmSimple.php
Opérations sur la table FilmSimple
Titre Année Réalisateur Année naissance Action
Vertigo 1959 Hitchcok Alfred 1899 Modifier ce film
Alien 1979 Ridley Scott 1943 Modifier ce film
Psychose 1960 Alfred Hitchcock 1899 Modifier ce film
Kagemusha 1980 Akira Kurosawa 1910 Modifier ce film
Volte-face 1997 John Woo 1946 Modifier ce film
Titanic 1997 James Cameron 1954 Modifier ce film
Sacrifice 1986 Andrei Tarkovski 1932 Modifier ce film
Table FilmSimple
Ajouter un film

Figure 2.5 — Page de mise à jour des films

Exemple 2.19 *exemples/FilmSimple.php* : *Script de gestion de la table FilmSimple*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<head>
<title>Opérations sur la table FilmSimple</title>
<link rel='stylesheet' href="films.css" type="text/css" />
</head>
<body>
```

```
<h2>Opérations sur la table <i>FilmSimple</i></h2>

<?php

require_once ("UtilFilmSimple.php");

// On normalise les entrées HTTP
Normalisation();

// Tableau "vide" utilisé comme valeurs par défaut pour les
// insertions
$NULL_FILM = array("titre" => "", "annee"=>"","nom_realisateur"=>"",
                   "annee_naissance"=>"","prenom_realisateur"=>"");

$connexion = Connexion (NOM, PASSE, BASE, SERVEUR);

if ( !isSet($_POST['action']) and !isSet($_GET['mode']) ) {
  // L'exécution n'est pas lancée depuis le formulaire
  //  ou depuis l'une des ancres créées dans TableauFilms()
  //  donc on affiche le tableau des films.

  TableauFilms ($connexion);
  // On place une ancre pour ajouter un film
  echo "<a href='FilmSimple.php?mode=" . MODE_INSERTION
  . "'>Ajouter un film</a>\n";
}
else {
  // Traitement des événements utilisateurs recueillis par
  // l'application

  if ( isSet($_GET['mode'])) {
    // L'utilisateur a cliqué l'une des ancres permettant de
    // modifier
    // ou d'ajouter un film
    if ($_GET['mode'] == MODE_MAJ) {
      // On récupère les données du film à modifier et on affiche
      // le formulaire pré-rempli à l'aide de ces données.

      $slash_titre = mysql_real_escape_string($_GET['titre']);
      $requete = "SELECT * FROM FilmSimple WHERE titre='
          $slash_titre'";
      $resultat = ExecRequete ($requete, $connexion);
      $film = LigneSuivante ($resultat);
      FormFilmSimple (MODE_MAJ, $film);
    }
    else if ($_GET['mode'] == MODE_INSERTION) {
      // On affiche un formulaire de saisie vierge
      FormFilmSimple (MODE_INSERTION, $NULL_FILM);
    }
  }
  else if (isSet($_POST['action'])) {
    // L'utilisateur a saisi des données dans le formulaire pour
```

```
      // modifier ou insérer un film, puis a cliqué sur "Exécuter"
      // On contrôle la saisie, met à jour la base et affiche
      // le tableau actualisé des films.

      // Contrôle des données
      if (ControleFilm ($_POST)) {
        MAJFilmSimple($_POST['mode'], $_POST, $connexion);
        TableauFilms ($connexion);
      }
    }
}
?>
</body>
</html>
```

L'affichage est déterminé par le paramètre `$mode` (qui indique dans quel mode on doit afficher le formulaire) et par le paramètre `$action` qui, quand il est présent, indique que le formulaire a été soumis.

Quand ces paramètres sont absents, on affiche simplement le tableau et l'ancre d'insertion. Quand `$mode` vaut `MODE_INSERTION`, c'est qu'on a utilisé l'ancre d'insertion : on appelle le formulaire en lui passant un tableau des valeurs par défaut vide. Quand `$mode` vaut `MODE_MAJ`, on sait qu'on reçoit également le titre du film à modifier : on le recherche dans la base et on passe le tableau obtenu à la fonction `FormFilmSimple()` pour tenir lieu de valeurs par défaut.

La variable `$action`, si elle est définie, indique qu'une mise à jour avec le formulaire a été effectuée. On effectue d'abord différents contrôles avec la fonction `ControleFilmSimple()` que nous détaillerons par la suite. Si cette fonction renvoie `true`, ce qui indique qu'il n'y a pas de problèmes, on appelle la fonction `MAJFilmSimple()` en lui passant le tableau des valeurs provenant du formulaire.

Le système obtenu permet d'effectuer des mises à jour (insertions et modifications) en suivant uniquement des URL dans lesquelles on a placé les informations décrivant l'action à effectuer. Il est très représentatif des techniques utilisées couramment pour accéder aux informations dans une base de données et les modifier. L'utilisation des fonctions permet de conserver un code relativement concis, dans lequel chaque action (mise à jour, affichage, contrôle) est bien identifiée et codée une seule fois. Sur le même principe, il est facile d'ajouter, dans le tableau HTML des films, une ancre pour détruire le film. Nous laissons cette évolution au lecteur, à titre d'exercice.

FilmSimple.php illustre encore une technique assez courante consistant à utiliser un seul script dans lequel des opérations différentes sont déclenchées en fonction de l'action précédemment effectuée par l'utilisateur. Cette technique peut être assez troublante dans un premier temps puisqu'elle nécessite de se représenter correctement la succession des interactions client/serveur pouvant mener à un état donné. Elle s'avère en pratique très utile, en évitant d'avoir à multiplier le nombre de scripts traitant d'une fonctionnalité bien identifiée.

Le point faible est la production du formulaire avec valeurs par défaut, assez lourde et qui le serait bien plus encore s'il fallait gérer de cette manière les listes déroulantes. Nous verrons dans le chapitre consacré à la programmation objet comment développer des outils automatisant dans une large mesure ce genre de tâche.

2.3.2 Validation des données et expressions régulières

Revenons une nouvelle et dernière fois sur les contrôles à effectuer lors de la réception des données soumises via un formulaire (voir page 70 pour une première approche). On peut effectuer des contrôles du côté client, avec JavaScript, ou du côté serveur, en PHP. Les contrôles JavaScript sont les plus agréables pour l'utilisateur puisqu'il n'a pas besoin d'attendre d'avoir saisi toutes ses données dans le formulaire et de les avoir transmises au serveur pour prendre connaissance des éventuels messages d'erreurs. En revanche la programmation JavaScript n'est pas une garantie puisqu'un esprit malfaisant peut très bien supprimer les contrôles avant de transmettre des informations à votre script. Il est donc indispensable d'ajouter dans tous les cas une validation des données côté serveur.

Les exemples qui suivent donnent quelques exemples de contrôles plus avancés que ceux donnés page 70. Nous prenons comme cas d'école la fonction de contrôle `ControleFilmSimple()` qui doit vérifier la validité des données avant insertion ou mise à jour de la table *FilmSimple* (voir ce qui précède). Voici tout d'abord la structure de cette fonction :

```
function ControleFilm ($film)
{
  // Ici des contrôles. Si une erreur est rencontrée, la variable
  // $message est définie

  ...

  // Fin des contrôles, affichage éventuel de $message

  if ($message) {
    echo "<b>Erreurs rencontrées :</b><br/>$message";
    FormFilmSimple (MODE_INSERTION, $film);
    return false;
  }
  else return true;
}
```

On prend donc en argument les données à placer dans la table (le tableau associatif `$film`) et on vérifie que tout est correct. Comment faire si une erreur a été rencontrée ? Une solution brutale est d'afficher le message et de redemander à l'utilisateur *toute* la saisie. Il faut s'attendre à une grosse colère de sa part s'il doit saisir à nouveau 20 champs de formulaire pour une erreur sur un seul d'entre eux.

La solution adoptée ici (et recommandée dans tous les cas) est de réafficher le formulaire avec les données saisies, en donnant également le message indiquant où la correction doit être faite. Il suffit bien sûr de reprendre la fonction

`FormFilmSimple()` (voilà qui devrait vous convaincre de l'utilité des fonctions ?) en lui passant le tableau des valeurs.

Suivent quelques contrôles possibles.

Existence, type d'une variable, présence d'une valeur

Rappelons tout d'abord comment vérifier l'existence des variables attendues. En principe le tableau `$film` doit contenir les éléments `titre`, `annee`, `prenom_realisateur`, `nom_realisateur` et `annee_naissance`. C'est toujours le cas si les données proviennent de notre formulaire de saisie, mais comme rien ne le garantit il faut tester l'existence de ces variables avec la fonction `isSet()`.

```
if (!isSet ($film['titre']))
    $message = "Pourquoi n'y a-t-il pas de titre ???<br/>";
```

Il faut également penser à vérifier que l'utilisateur a bien saisi un champ. Voici le test pour le nom du metteur en scène (l'expression « `$a .= $b` » est un abrégé pour « `$a = $a . $b` »).

```
if (empty($film['nom_realisateur']))
   $message .= "Vous devez saisir le nom du metteur en scène<br/>
       ";
```

Si les variables existent, on peut tester le type avec les fonctions PHP `is_string()`, `is_numeric()`, `is_float()`, etc. (voir annexe C). Voici comment tester que l'année est bien un nombre.

```
if (!is_numeric ($film['annee']))
  $message = $message . "L'année doit être un entier : $film[
      annee]<br/>";
```

Les tests sur le contenu même de la variable sont d'une très grande variété et dépendent fortement de l'application. On pourrait vérifier par exemple que l'année a une valeur raisonnable, que le nom et le prénom débutent par une capitale, ne contiennent pas de caractère blanc en début de chaîne, ne contiennent pas de chiffre, que le metteur en scène est né avant de réaliser le film (!), etc.

Une partie des tests, celle qui concerne le *format* des valeurs saisies, peut s'effectuer par des expressions régulières.

Validation par expressions régulières

Les expressions régulières[4] permettent de définir des « *patterns* », ou motifs, que l'on peut ensuite rechercher dans une chaîne de caractères (*pattern matching*). Un exemple très simple, déjà rencontré, est le test d'une occurrence d'une sous-chaîne dans une chaîne avec l'opérateur `LIKE` de SQL. La requête suivante sélectionne ainsi tous les films dont le titre contient la sous-chaîne « ver ».

```
SELECT * FROM FilmSimple WHERE titre LIKE '%ver%'
```

4. On parle aussi d'expressions rationnelles.

Les expressions régulières autorisent une recherche par motif beaucoup plus puissante. Une expression décrit un motif en indiquant d'une part le caractère ou la sous-chaîne attendu(e) dans la chaîne, et en spécifiant d'autre part dans quel ordre et avec quel nombre d'occurrences ces caractères ou sous-chaînes peuvent apparaître.

L'expression régulière la plus simple est celle qui représente une sous-chaîne constante comme, par exemple, le « ver » dans ce qui précède. Une recherche avec cette expression a la même signification que la requête SQL ci-dessus. Il est possible d'indiquer plus précisément la place à laquelle doit figurer le motif :

1. le « ^ » indique le début de la chaîne : l'expression « ^ver » s'applique donc à toutes les chaînes commençant par « ver » ;
2. le $ indique la fin de la chaîne : l'expression « ver$ » s'applique donc à toutes les chaînes finissant par « ver » ;

On peut exprimer de manière concise toute une famille de motifs en utilisant les symboles d'occurrence suivants :

1. « `m*` » indique que le motif `m` doit être présent 0 ou plusieurs fois ;
2. « `m+` » indique que le motif `m` oit être présent une (au moins) ou plusieurs fois, ce qu'on pourrait également exprimer par « mm* » ;
3. « `m?` » indique que le motif `m` peut être présent 0 ou une fois ;
4. « `m{p,q}` » indique que le motif `m` peut être présent au moins `p` fois et au plus `q` fois (la syntaxe `{p,}` indique simplement le « au moins », sans maximum, et `{p}` est équivalent à`{p,p}`).

Par défaut, les symboles d'occurrence s'appliquent au caractère qui précède, mais on peut généraliser le mécanisme avec les parenthèses qui permettent de créer des séquences. Ainsi `(ver)+` est une expression qui s'applique aux chaînes contenant au moins une fois la sous-chaîne `ver`, alors que `ver+` s'applique aux sous-chaînes qui contiennent `ve` suivi d'un ou plusieurs `r`.

Le choix entre plusieurs motifs peut être indiqué avec le caractère « | ». Par exemple l'expression `ver+|lie+` s'applique aux chaînes qui contiennent au moins une fois `ver` *ou* au moins une fois `lie`. Pour vérifier qu'une chaîne contient un chiffre, on peut utiliser l'expression `0|1|2|3|4|5|6|7|8|9` mais on peut également encadrer tous les caractères acceptés entre crochets : `[0123456789]`. Une expression constituée d'un ensemble de caractères entre crochets s'applique à toutes les chaînes contenant au moins un de ces caractères. Si le premier caractère entre les crochets est ^, l'interprétation est inversée : l'expression s'applique à toutes les chaînes qui *ne contiennent pas* un des caractères. Voici quelques exemples :

- `[ver]` : toutes les chaînes avec un v, un e ou un `r` ;
- `[a-f]` : toutes les chaînes avec une des lettres entre a et f ;

- `[a-zA-Z]` : toutes les chaînes avec une lettre de l'alphabet.
- `[^0-9]` : toutes les chaînes sans chiffre.

Pour simplifier l'écriture des expressions certains mot-clés représentent des classes courantes de caractères, données dans la table 2.2. Ils doivent apparaître dans une expression régulière encadrés par « : » pour éviter toute ambiguité comme, par exemple, « `:alpha:` ».

Tableau 2.2 — Classes de caractères

Mot-clé	Description
`alpha`	N'importe quel caractère alphanumérique.
`blank`	Espaces et tabulations.
`cntrl`	Tous les caractères ayant une valeur ASCII inférieure à 32.
`lower`	Toutes les minuscules.
`upper`	Toutes les majuscules.
`space`	Espaces, tabulations et retours à la ligne.
`xdigit`	Chiffres en hexadécimal.

Enfin le point « . » représente n'importe quel caractère, sauf le saut de ligne `NEWLINE`. Le point, comme tous les caractères spéciaux (`^, ., [, ], (, ), *, +, ?, {, , }, \`) doit être précédé par un \ pour être pris en compte de manière littérale dans une expression régulière.

Expressions régulières et PHP

Les deux principales fonctions PHP pour traiter des expressions régulières sont `ereg()` et `ereg_replace()`. La première prend trois arguments : l'expression régulière, la chaîne à laquelle on souhaite appliquer l'expression, enfin le dernier paramètre (optionnel) est un tableau dans lequel la fonction placera toutes les occurrences de motifs, rencontrés dans la chaîne, satisfaisant l'expression régulière.

Voici un exemple pour notre fonction de contrôle. On veut tester si l'utilisateur place des balises dans les chaînes de caractères, notamment pour éviter des problèmes à l'affichage. Voyons d'abord l'expression représentant une balise. Il s'agit de toute chaîne commençant par « < », suivi de caractères à l'exception de « > », et se terminant par « > ». L'expression représentant une balise est donc

```
<[^>]*>
```

Voici le test appliqué au nom du metteur en scène :

```
if (ereg ("<[^>]*>", $film['nom_realisateur'], $balises))
      $message .= "Le nom contient la balise : "
                  . htmlEntities ($balises[0]);
```

La fonction `ereg()` recherche dans le nom toutes les balises, et les place dans le tableau `$balises`. Elle renvoie `true` si au moins un motif a été trouvé dans la chaîne.

On donne alors dans le message d'erreur la balise rencontrée (on pourrait les afficher toutes avec une boucle). Attention : pour qu'une balise apparaisse textuellement dans la fenêtre d'un navigateur, il faut l'écrire sous la forme `<balise>` pour éviter qu'elle ne soit interprétée comme une directive de mise en forme. La fonction `htmlEntities()` remplace dans une chaîne tous les caractères non-normalisés d'un texte HTML par l'entité HTML correspondante.

Voici un autre exemple testant que le nom du metteur en scène ne contient que des caractères alphabétiques. Si on trouve un tel caractère, on le remplace par une « * » avec la fonction `ereg_replace()` afin de marquer son emplacement.

```
if (ereg ("[^A–Za–z]", $film['nom_realisateur']))
    $message .= "Le nom contient un ou plusieurs caractères "
              . "  non–alphabétiques : "
              . ereg_replace ("[^A–Za–z]", "*", $film['
                nom_realisateur'])
              . "<br/>";
```

La fonction `ereg_replace()` a pour but de remplacer les motifs trouvés dans la chaîne (troisième argument) par une sous-chaîne donnée dans le second argument.

Les expressions régulières sont indispensables pour valider toutes les chaînes dont le format est contraint, comme les nombres, les unités monétaires, les adresses électroniques ou HTTP, etc. Elles sont également fréquemment utilisées pour inspecter la variable `USER_AGENT` et tester le navigateur utilisé par le client afin d'adapter l'affichage aux particularités de ce navigateur (voir également la fonction PHP `get_browser()`).

2.4 TRANSFERT ET GESTION DE FICHIERS

Nous montrons maintenant comment échanger des fichiers de type quelconque entre le client et le serveur. L'exemple pris est celui d'un album photo en ligne (très limité) dans lequel l'internaute peut envoyer des photos stockées sur le serveur avec une petite description, consulter la liste des photos et en récupérer certaines. La première chose à faire est de vérifier que les transferts de fichier sont autorisés dans la configuration courante de PHP. Cette autorisation est configurée par la directive suivante dans le fichier *php.ini* :

```
; Whether to allow HTTP file uploads.
file_uploads = On
```

Une seule table suffira pour notre application. Voici le script de création.

Exemple 2.20 *exemples/Album.sql* : *Table pour l'album photos*

```
# Création d'une table pour un petit album photo

CREATE TABLE Album
    (id INTEGER AUTO_INCREMENT NOT NULL,
    description TEXT,
```

```
    compteur INTEGER DEFAULT 0,
    PRIMARY KEY (id)
)
;
```

L'attribut `compteur`, de valeur par défaut 0, donnera le nombre de téléchargements de chaque photo.

2.4.1 Transfert du client au serveur

Voici le formulaire permettant d'entrer le descriptif et de choisir le fichier à transmettre au serveur. Il est très important, pour les formulaires transmettant des fichiers, d'utiliser le mode `post` et de penser à placer l'attribut `enctype` à la valeur `multipart/form-data` dans la balise <**form**> (voir chapitre 1, page 10). Le transfert d'un fichier donne en effet lieu à un message HTTP en plusieurs parties. En cas d'oubli de cet attribut le fichier n'est pas transmis.

Rappelons que les champs <**input**> de type `file` créent un bouton de formulaire permettant de parcourir les arborescences du disque local pour choisir un fichier. Dans notre formulaire, ce champ est nommé `maPhoto`. Notez le champ caché `max_file_size` qui limite la taille du fichier à transférer.

Exemple 2.21 *exemples/FormTransfert.html* : *Formulaire pour sélectionner le fichier à transmettre*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Formulaire de transfert de photographie</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>Transfert de photographies dans l'album</h1>

<form  enctype="multipart/form-data" action="TransfertFichier.php
    "
       method='post'>
  <p>
    <textarea name="description"
          cols='50' rows='3'>Entrez ici la description de la
              photographie
    </textarea>
  </p><p>
   Choisissez le fichier : <br/>
    <input type='hidden' name='max_file_size' value='2000000'/>
    <input type='file' size='40' name='maPhoto'/>
```

```
  </p>

  <input type='submit' value='Transférer'/>
</form>
</body>
</html>
```

Voici maintenant le script *TransfertFichier.php* associé à ce formulaire, montrant comment le fichier est traité à l'arrivée sur le serveur. L'instruction `switch`, analogue à celle du C ou du C++, permet de choisir une action à effectuer en fonction de la valeur d'une variable (ici `$codeErreur`) ou d'une expression : voir page 431 pour plus de détails.

Les informations relatives aux fichiers transférés sont disponibles dans un tableau `$_FILES` à deux dimensions. La première est le nom du champ de formulaire d'où provient le fichier (dans notre cas, `maPhoto`) ; la seconde est un ensemble de propriétés décrivant le fichier reçu par le serveur, énumérées dans la table 2.3. La propriété `error` permet de savoir si le transfert s'est bien passé ou, si ce n'est pas le cas, quel type de problème est survenu. Les valeurs possibles du code d'erreur sont les suivantes :

- `UPLOAD_ERR_OK` : pas d'erreur, le transfert s'est bien effectué ;
- `UPLOAD_ERR_INI_SIZE` : le fichier transmis dépasse la taille maximale autorisée, cette dernière étant paramétrée dans le fichier *php.ini* :

  ```
  ; Maximum allowed size for uploaded files.
  upload_max_filesize = 2M
  ```

- `UPLOAD_ERR_FORM_SIZE` : la taille du fichier dépasse celle indiquée dans la directive `max_file_size` qui peut être spécifiée dans le formulaire HTML ;
- `UPLOAD_ERR_PARTIAL` : le fichier a été transféré seulement partiellement ;
- `UPLOAD_ERR_NO_FILE` : aucun fichier n'a été transféré.

Tableau 2.3 — Variables décrivant un transfert de fichier (seconde dimension du tableau `$_FILES`)

Nom	Description
`name`	Nom du fichier sur la machine du client.
`tmp_name`	Nom du fichier temporaire sur la machine du serveur.
`size`	Taille du fichier, en octets.
`type`	Le type MIME du fichier, par exemple « image/gif »
`error`	Code d'erreur si le fichier n'a pu être transmis correctement (depuis la version 4.2 de PHP).

Exemple 2.22 *exemples/TransfertFichier.php* : *Script de traitement du fichier*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
```

```
<head>
<title>Transfert du fichier</title>
<link rel='stylesheet' href="films.css" type="text/css" />
</head>
<body>

<h1>Réception du fichier</h1>

<?php
require_once ("Connect.php");
require_once ("Connexion.php");
require_once ("ExecRequete.php");
require_once ("Normalisation.php");

// Normalisation des données HTTP
Normalisation();

// Récupération du code indicateur du transfert
$codeErreur = $_FILES['maPhoto']['error'];

if ($codeErreur == UPLOAD_ERR_OK) {
  // Le fichier a bien été transmis
  $fichier = $_FILES['maPhoto'];
  echo "<b>Nom du fichier client :</b> " . $fichier['name'] .
       "<br/>";
  echo "<b>Nom du fichier serveur :</b> " .$fichier['tmp_name'] .
       "<br/>";
  echo "<b>Taille du fichier :</b> " . $fichier['size'] . "<br/>";
  echo "<b>Type du fichier :</b>" . $fichier['type']  . "<br/>";

  // On insère la description dans la table Album

  $connexion = Connexion (NOM, PASSE, BASE, SERVEUR);
  // Protection des données à insérer
  $description =
  htmlSpecialChars(mysql_real_escape_string($_POST
                                            ['description']));
  $requete = "INSERT INTO Album (description) VALUES
                                            ('$description')";

  $resultat = ExecRequete ($requete, $connexion);

  // On récupère l'identifiant attribué par MySQL
  $id = mysql_insert_id ($connexion);

  // Copie du fichier dans le répertoire PHOTOS
  copy($fichier['tmp_name'], "./PHOTOS/$id.jpg");

}
else {
  // Une erreur quelque part
  switch ($codeErreur)  {
```

```
      case UPLOAD_ERR_NO_FILE:
        echo "Vous avez oublié de transmettre le fichier !?\n";
        break;

      case UPLOAD_ERR_INI_SIZE:
        echo "Le fichier dépasse la taille max. autorisée par PHP";
        break;

      case UPLOAD_ERR_FORM_SIZE:
        echo "Le fichier dépasse la taille max. autorisée par le
            formulaire";
        break;

      case UPLOAD_ERR_PARTIAL:
        echo "Le fichier a été transféré partiellement";
        break;

      default:
        echo "Ne doit pas arriver!!!";
    }
}
?>
</body>
</html>
```

Le script teste soigneusement ces erreurs et affiche un message approprié au cas de figure. Si un fichier est transféré correctement sur le serveur, ce dernier le copie dans un répertoire temporaire (sous Unix, `/tmp`, paramétrable dans le fichier de configuration *php.ini*). Le nom de ce fichier temporaire est donné (dans notre cas) par `$_FILES['maPhoto']['tmp_name']`. Notre script affiche alors les quatre informations connues sur ce fichier.

On insère ensuite la description du fichier dans la table *Album*. Remarquez que l'attribut `id` n'est pas donné dans la commande `INSERT` : MySQL se charge d'attribuer automatiquement un identifiant aux champs `AUTO_INCREMENT`. La fonction `mysql_insert_id()` permet de récupérer l'identifiant attribué par le dernier ordre `INSERT` effectué.

Finalement, on copie (fonction `copy()`) le fichier temporaire dans le sous-répertoire *PHOTOS*, en lui donnant comme nom l'identifiant de la description dans MySQL, et comme extension « .jpg ». On a supposé ici pour simplifier que tous les fichiers transmis sont au format JPEG, mais il serait bon bien sûr de choisir l'extension en fonction du type MIME du fichier transmis.

Attention : le processus qui effectue cette copie est le programme serveur. Ce programme doit impérativement avoir les droits d'accès et d'écriture sur les répertoires dans lesquels on copie les fichiers (ici c'est le répertoire `PHOTOS` situé sous le répértoire contenant le script *TransfertFichier.php*).

Quelques précautions

Le transfert de fichiers extérieurs sur une machine serveur nécessite quelques précautions. Il est évidemment très dangereux d'exécuter un script PHP ou un programme reçu via le Web. Faites attention également à ne pas permettre à celui qui transfère le fichier d'indiquer un chemin d'accès absolu sur la machine (risque d'accès à des ressources sensibles). Enfin il est recommandé de contrôler la taille du fichier transmis, soit au niveau du navigateur, soit au niveau du serveur.

Du côté navigateur, un champ caché de nom `max_file_size` peut précéder le champ de type `file` (voir l'exemple de *FormTransfert.html* ci-dessus). Le navigateur doit alors en principe interdire le transfert de fichier plus gros que la taille maximale indiquée. Comme tous les contrôles côté client, il ne s'agit pas d'une garantie absolue, et il est préférable de la doubler côté serveur. Le paramètre `upload_max_filesize` dans le fichier *php.ini* indique à PHP la taille maximale des fichiers recevables.

Le transfert de fichiers sur le serveur peut être dangereux, et mérite que vous consacriez du temps et de la réflexion à vérifier que la sécurité n'est pas compromise. Tenez-vous également au courant des faiblesses détectées dans les différentes versions de PHP en lisant régulièrement les informations sur le sujet publiées dans les forums spécialisés ou sur *http://www.php.net*.

2.4.2 Transfert du serveur au client

Voyons maintenant comment transférer des fichiers du serveur au client. À titre d'illustration, nous allons afficher la liste des photos disponibles et proposer leur téléchargement.

En principe il existe autant de lignes dans la table *Album* que de fichiers dans le répertoire *PHOTOS*. On peut donc, au choix, parcourir la table *Album* et accéder, pour chaque ligne, au fichier correspondant, ou l'inverse. Pour les besoins de l'exemple, nous allons adopter ici la seconde solution.

PHP propose de nombreuses fonctions pour lire, créer, supprimer ou modifier des fichiers et des répertoires. Nous utilisons ici les fonctions `opendir()` , qui renvoie un identifiant permettant d'accéder à un répertoire, `readir()` qui permet de parcourir les fichiers d'un répertoire, et `closedir()` qui ferme l'accès à un répertoire. Voici ces fonctions à l'œuvre dans le script *ListePhotos.php*.

Exemple 2.23 *exemples/ListePhotos.php* : *Affichage de la liste des photos*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<head>
<title>Liste et téléchargement des photos</title>
<link rel='stylesheet' href="films.css" type="text/css" />
</head>
```

```
<body>

<h1>Liste et téléchargement des photos</h1>

<?php
require_once ("UtilBD.php");

$connexion = Connexion (NOM, PASSE, BASE, SERVEUR);

// On affiche la liste des photos

echo "<table border='4' cellspacing='2' cellpadding='2'>"
. "<caption ALIGN='bottom'>Les photos disponibles</caption> "
. "<tr><th>Vignette</th><th>Description</th><th>Taille</th>"
. "<th>Agrandir</th><th>Compteur</th><th>Action</th></tr>\n";

$dir = opendir ("PHOTOS");
while ($fichier = readdir($dir)) {
  if (ereg ("\.jpg\$", $fichier)) {
    $id = substr ($fichier, 0, strpos ($fichier, "."));
    $requete = "SELECT * FROM Album WHERE id='$id'";
    $resultat = ExecRequete ($requete, $connexion);
    $photo = ObjetSuivant($resultat);

    echo "<tr><td><img src='PHOTOS/$fichier' height='70' width
        ='70'/></td>"
    . "<td>$photo->description</td>"
    . "<td>" . filesize("PHOTOS/$fichier") . "</td>"
    . "<td><a href='PHOTOS/$fichier'>$fichier</a></td>"
    . "<td>$photo->compteur</td>"
    . "<td><a href='ChargerPhoto.php?id=$id'> "
    .     "Télécharger cette photo</a></td>\n";
  }
}
echo "</table>\n";
closedir ($dir);

?>
<a href='FormTransfert.html'>Ajouter une photo</a>
</body>
</html>
```

L'accès aux fichiers du répertoire se fait avec la boucle suivante :

```
$dir = opendir ("PHOTOS");
while ($fichier = readdir($dir)) {
 // On ne prend que les fichiers JPEG
 if (ereg ("\.jpg\$", $fichier)) {
 }
}
closedir ($dir);
```

À l'intérieur de la boucle on veille à ne prendre que les fichiers JPEG en testant avec une expression régulière (voir section précédente) que l'extension est bien « jpg ». Notez que le caractère réservé PHP « $ » est précédé de \ pour s'assurer qu'il est bien passé littéralement à la fonction `ereg()`, où il indique que la chaîne doit se terminer par « jpeg ». Le point, « . » est lui un caractère réservé dans les expressions régulières (il représente n'importe quel caractère), et on « l'échappe » donc également pour qu'il soit interprété littéralement.

On récupère ensuite l'identifiant de la photographie en prenant, dans le nom du fichier, la sous-chaîne précédant le « . », dont on se sert pour chercher la description et le compteur de la photographie dans la base. Il ne reste plus qu'à afficher une ligne du tableau HTML.

- Pour afficher une vignette (format réduit) de la photo, on utilise la balise <**img**> en indiquant une hauteur et une largeur limitée à 70 pixels.
- On peut accéder à l'image complète avec l'ancre qui fait référence au fichier JPEG. Si l'utilisateur choisit cette ancre, le serveur envoie automatiquement un fichier avec un en-tête `image/jpeg` qui indique au navigateur qu'il s'agit d'une image et pas d'un fichier HTML.
- Enfin, la fonction `filesize()` renvoie la taille du fichier passée en argument.

Le téléchargement du fichier image nous montre pour conclure comment compter le nombre d'accès à un fichier. Il y a deux problèmes à résoudre. Le premier est « d'intercepter » la demande de téléchargement pour pouvoir exécuter l'ordre SQL qui va incrémenter le compteur. Le second est d'éviter que le fichier s'affiche purement et simplement dans la fenêtre du navigateur, ce qui n'est pas le but recherché. Il faut au contraire que, quel que soit le type du fichier transmis, le navigateur, au lieu de l'afficher, propose une petite fenêtre demandant dans quel répertoire de la machine client on doit le stocker, et sous quel nom.

La solution consiste à utiliser un script intermédiaire, *ChargerPhoto.php* qui, contrairement à tout ceux que nous avons vus jusqu'à présent, ne produit aucune ligne HTML. Ce script nous permet d'intercaler l'exécution d'instructions PHP entre la demande de l'utilisateur et la transmission du fichier. On peut donc résoudre facilement les deux problèmes précédents :

1. l'identifiant du fichier à récupérer est passé au script en mode `get` (voir le script *ListePhotos.php* ci-dessus) : on peut donc incrémenter le compteur dans la table *Album* ;
2. on donne explicitement l'en-tête du fichier transmis grâce à la fonction `Header()`.

Exemple 2.24 *exemples/ChargerPhoto.php* : *Script de téléchargement d'une photo*

```
<?php
  require_once ("Connect.php");
  require_once ("Connexion.php");
  require_once ("ExecRequete.php");
```

```
// Téléchargement d'une photo identifiée par $_GET['id']

// On commence par incrémenter le compteur

$connexion = Connexion (NOM, PASSE, BASE, SERVEUR);
$requete =  "UPDATE Album SET compteur=compteur+1 "
          . "WHERE id='{$_GET['id']}'";
$resultat = ExecRequete ($requete, $connexion);

// On envoie un en-tête forçant le transfert (download)
$fichier = $_GET['id'] . ".jpg";
$chemin = "PHOTOS/" ;
header ("Content-type: application/force-download");
header ("Content-disposition: filename=$fichier");

// Après l'en-tête on transmet le contenu du fichier lui-même
readFile ($chemin . $fichier);

?>
```

L'incrémentation du compteur est totalement transparente pour l'utilisateur. L'information `Content-type` de l'en-tête demande au navigateur de traiter le contenu du message HTTP comme un fichier à stocker, tandis que `Content-disposition` permet de proposer un nom par défaut, dans la fenêtre de téléchargement, pour ce fichier. Enfin, la fonction `readfile()` ouvre un fichier et transfère directement son contenu au navigateur.

L'intérêt de ce genre de script est de permettre d'exécuter un traitement quelconque en PHP, sans aucun affichage, puis de renvoyer à une autre URL sans que l'utilisateur ait à intervenir. On pourrait ici, en plus de l'incrémentation du compteur, regarder qui vient chercher une image (en inspectant son adresse IP, disponible dans la variable serveur `REMOTE_ADDR`, ou toute autre information contenue dans les variables CGI, voir page 16).

L'en-tête `Location: autreURL` par exemple permet de renvoyer à l'URL `autreURL`, qui peut être un script PHP ou un fichier HTML produisant réellement l'affichage.

2.5 SESSIONS

Comme nous l'avons déjà évoqué dans le chapitre 1, le protocole HTTP ne conserve pas d'informations sur la communication entre un programme client et un programme serveur. Le terme de *session* web désigne les mécanismes qui permettent d'établir une certaine continuité dans les échanges entre un client et un serveur donnés. Ces mécanismes ont en commun l'attribution d'un *identifiant de session* à un utilisateur, et la mise en œuvre d'un système permettant de transmettre systématiquement l'identifiant de session à chaque accès du navigateur vers le serveur. Le serveur sait alors à qui il a affaire, et peut accumuler de l'information sur cet interlocuteur.

2.5.1 Comment gérer une session web ?

Il existe essentiellement deux systèmes possibles. Le premier consiste à insérer l'identifiant de session dans *toutes* les URL des pages transmises au client, ainsi que dans *tous* ses formulaires. Cette solution, conforme au standard HTTP, est très lourde à mettre en œuvre. De plus elle s'avère très fragile puisqu'il suffit que l'internaute accède à ne serait-ce *qu'une* seule page d'un autre site pour que l'identifiant de session soit perdu.

La deuxième solution est de créer un ou plusieurs *cookies* pour stocker l'identifiant de session, (et peut-être d'autres informations) du côté du programme client. Rappelons (voir la fin du chapitre 1, page 16) qu'un *cookie* est essentiellement une donnée transmise par le programme serveur au programme client, ce dernier étant chargé de la conserver pour une durée déterminée. Cette durée peut d'ailleurs excéder la durée d'exécution du programme client lui-même, ce qui implique que les cookies soient stockés dans un fichier texte sur la machine cliente.

On peut créer des *cookies* à partir d'une application PHP avec la fonction `SetCookie()`, placée dans l'en-tête du message HTTP. Dès que le navigateur a reçu (et accepté) un cookie venant d'un serveur, il renvoie ce *cookie* avec tous les accès à ce même serveur, et ce durant toute la durée de vie du *cookie*. Ce processus est relativement sécurisé puisque seul le programme serveur qui a créé un *cookie* peut y accéder, ce qui garantit qu'un autre serveur ne peut pas s'emparer de ces informations. En revanche, toute personne pouvant lire des fichiers sur la machine cliente peut alors trouver les *cookies* en clair dans le fichier *cookies*.

Les *cookies* ne font pas partie du protocole HTTP, mais ont justement été inventés pour pallier les insuffisances de ce dernier. Ils sont reconnus par tous les navigateurs, même si certains proposent à leurs utilisateurs de refuser d'enregistrer les *cookies* sur la machine.

Voici un script PHP qui montre comment gérer un compteur d'accès au site avec un *cookie*.

Exemple 2.25 *exemples/SetCookie.php* : *Un compteur d'accès réalisé avec un cookie*

```
<?php
// Est-ce que le cookie existe ?
if (isSet($_COOKIE['compteur'])) {
    $message = "Vous êtes déjà venu {$_COOKIE['compteur']} fois "
      . "me rendre visite<br/>\n";
    // On incrémente le compteur

    $valeur = $_COOKIE['compteur'] + 1;
}
else {
    // Il faut créer le cookie avec la valeur 1
    $message = "Bonjour, je vous envoie un cookie<br/>\n";
    $valeur = 1;
}
```

```
// Envoi du cookie
SetCookie ("compteur", $valeur);

?>
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Les cookies</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>

<body>

<h1>Un compteur d'accès au site avec cookie</h1>

<?php echo $message; ?>

</body>
</html>
```

L'affichage se limite à un message qui varie selon que c'est la première fois ou non qu'un utilisateur se connecte au site. À chaque passage (essayez de recharger plusieurs fois la page) le compteur stocké dans le *cookie* est récupéré et incrémenté. Ces instructions sont placées avant toute sortie HTML puisque la définition d'un *cookie* fait partie de l'en-tête HTTP. Si l'on commet l'erreur de transmettre ne serait-ce qu'un caractère blanc avant le cookie, on obtient le message suivant:

```
Warning: Cannot modify header information -
headers already sent by (output started
       at /Applications/MAMP/htdocs/exemples/SetCookie.php:2)
in /Applications/MAMP/htdocs/exemples/SetCookie.php on line 18
```

Dans ce cas regardez votre code à l'emplacement indiqué par le message, et cherchez les caractères transmis avant toute instruction plaçant quelque chose dans l'en-tête.

> **REMARQUE –** Dans les fichiers contenant des déclarations de fonctions ou de classes, une bonne habitude à prendre est de ne pas placer la balise fermante PHP. Cela ne gène pas l'interpréteur, tout en évitant d'introduire des caractères parasites après la balise fermante.

L'appel à `SetCookie()` crée le *cookie* la première fois, et modifie sa valeur les fois suivantes. Par défaut, la durée de vie de ce *cookie* est celle du processus client (le navigateur) mais il est possible de donner une date pour le garder plusieurs jours, mois ou années (voir page 16).

2.5.2 Gestion de session avec *cookies*

Voyons maintenant comment utiliser les *cookies* pour gérer des sessions et enregistrer des informations sur un internaute dans une base de données. Les étapes à mettre en œuvre sont les suivantes :

1. quand un internaute arrive pour la première fois sur le site, on lui attribue un *identifiant unique*, et on lui transmet cet identifiant dans un *cookie* ;
2. à chaque accès ultérieur on est capable de reconnaître l'internaute par son identifiant, et on peut mémoriser les informations le concernant dans une ou plusieurs tables ;
3. quand la session est terminée, on peut valider définitivement l'ensemble des actions effectuées par l'internaute, éventuellement en lui demandant confirmation.

L'exemple donné ci-dessous consiste à proposer un menu en plusieurs phases : les entrées, les plats, puis les desserts, en conservant à chaque fois l'information sur les choix précédents.

> **REMARQUE –** PHP propose un mécanisme pour gérer les sessions. Cependant, pour clarifier les choses, nous décrivons dans un premier temps une technique indépendante avant de montrer l'équivalent avec les fonctions PHP. Le chapitre 7 montre comment combiner sessions web et authentification d'accès à un site.

Voici la table *Carte* contenant les choix possibles, avec leur type.

Exemple 2.26 *exemples/Carte.sql* : *La carte du restaurant*

```
# Création d'une table pour la carte d'un restaurant

CREATE TABLE Carte
(id_choix       INTEGER AUTO_INCREMENT NOT NULL,
 libelle          TEXT,
 type          ENUM ("Entrée", "Plat", "Dessert"),
 PRIMARY KEY (id_choix)
);

INSERT INTO Carte(libelle , type) VALUES("Crudités", "Entrée");
INSERT INTO Carte(libelle , type) VALUES("Charcuterie", "Entrée");
INSERT INTO Carte(libelle , type) VALUES("Hareng", "Entrée");

INSERT INTO Carte(libelle , type) VALUES("Steak", "Plat");
INSERT INTO Carte(libelle , type) VALUES("Turbot", "Plat");
INSERT INTO Carte(libelle , type) VALUES("Choucroute", "Plat");

INSERT INTO Carte(libelle , type) VALUES("Paris-Brest", "Dessert");
INSERT INTO Carte(libelle , type) VALUES("Crème caramel", "Dessert");
INSERT INTO Carte(libelle , type) VALUES("Tarte citron", "Dessert");
```

Il nous faut une autre table pour conserver les choix d'un internaute. Cette table associe l'internaute représenté par son identifiant de session, et un choix de la carte représenté par son identifiant `id_choix`.

Exemple 2.27 *exemples/Commande.sql* : *Les commandes de l'internaute*

```
# Création d'une table pour les commandes au restaurant

CREATE TABLE Commande
(id_session VARCHAR (40) NOT NULL,
 id_choix INTEGER NOT NULL,
 PRIMARY KEY (id_session, id_choix)
);
```

Passons maintenant à la réalisation du système de commandes. Il faut d'abord prévoir une fonction pour afficher les choix de la carte en fonction du type (entrée, plat ou dessert). Ces choix sont proposés avec un bouton de type `radio`.

Exemple 2.28 *exemples/FormCommande.php* : *Le formulaire d'affichage d'un choix à la carte*

```
<?php
 // Formulaire de saisie d'un choix à la carte
 function FormCommande ($type_choix, $script, $connexion)
 {
   // Un message pour indiquer à quel stade on en est
   if ($type_choix == "Entrée")
    echo "Pour commencer nous vous proposons les entrées<br/>";
   else
   if ($type_choix == "Plat")
      echo "Maintenant choisissez un plat<br/>";
   else
      echo "Enfin choisissez un dessert<br/>";

   // Maintenant on crée le formulaire
   echo "<form action='$script' method='post'>\n";

   // Champ caché avec le type de choix
   echo "<input type='hidden' name='type_choix' value='
       $type_choix'/>";

   // Recherche des choix selon le type (entrée, plat ou dessert)
   $requete = "SELECT * FROM Carte WHERE type='$type_choix'";
   $resultat = ExecRequete ($requete, $connexion);

   // Affichage des choix
   while ($choix = ObjetSuivant ($resultat))
     echo "$choix->libelle : "
      . "<input type='radio' name='id_choix' value='$choix->
          id_choix'/><br/>";
```

```
    echo "<input type='submit' value='Exécuter'/>\n";
    echo "</form>\n";
  }
?>
```

La figure 2.6 montre le formulaire affiché par le script *ExSession.php*, avec les entrées. Voici le code de ce script.

Figure 2.6 — Le formulaire, au début de la session

Exemple 2.29 *exemples/ExSession.php* : *Exemple de commande avec session*

```
<?php
if (isSet($_COOKIE['id_session'])) {
  // L'identifiant de session existe déjà
  $id_session = $_COOKIE['id_session'];
}
else {
  // Créons un identifiant
  $id_session = $_SERVER['REMOTE_ADDR'] . date("U");

  // Envoi du cookie
  SetCookie ("id_session", $id_session);
}
?>
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<head>
```

```
<title>Une commande au restaurant</title>
<link rel='stylesheet' href="films.css" type="text/css" />
</head>
<body>

<h1>Faites votre commande au restaurant</h1>

<?php
require_once ("Connect.php");
require_once ("Connexion.php");
require_once ("ExecRequete.php");
require_once ("Normalisation.php");
require_once ("FormCommande.php");

// Connexion à la base
$connexion = Connexion (NOM, PASSE, BASE, SERVEUR);

// Normalisation des entrées HTTP
Normalisation();

// Si le type de choix n'est pas défini : on commence
// par proposer les entrées

if (!isSet($_POST['type_choix'])) {
  echo "Bonjour. Nous vous avons attribué la session $id_session
       <br/>";
  FormCommande ("Entrée", "ExSession.php", $connexion);
}
else {
  // Insérons dans la table le choix qui vient d'être fait
  // Il faudrait tester que id_choix est défini...
  $requete = "INSERT INTO Commande (id_session, id_choix) "
  . "VALUES ('$id_session', '{$_POST['id_choix']}')";
  ExecRequete ($requete, $connexion);

  // Affichage des choix déjà effectués
  $requete = "SELECT C2.* FROM Commande C1, Carte C2"
  . " WHERE id_session='$id_session' AND C1.id_choix=C2.id_choix"
  . " ORDER BY C2.id_choix ";
  $resultat = ExecRequete ($requete, $connexion);
  while ($choix = ObjetSuivant ($resultat))
  echo "Vous avez choisi : $choix->libelle<br/>\n";

  // Affichage de la suite
  if ($_POST['type_choix'] == 'Entrée') {
    FormCommande ("Plat", "ExSession.php", $connexion);
  }
  else if ($_POST['type_choix'] == 'Plat') {
    FormCommande ("Dessert", "ExSession.php", $connexion);
  }
  else {
    // Traitement de la commande complète. Ici on détruit...
```

```
    echo "Nous avons noté votre commande. Merci !<br/>";
    $requete = "DELETE FROM Commande WHERE id_session='
        $id_session'";
    ExecRequete ($requete, $connexion);
  }
}
?>
</body>
</html>
```

La première partie est relativement analogue à celle du premier exemple avec *cookies*, page 99. Si l'identifiant de session existe, on le conserve. Sinon on le calcule, et on crée le *cookie* pour le récupérer aux accès suivants. Pour l'identifiant, nous avons choisi ici simplement de concaténer l'adresse IP du client et le temps « Unix » lors de la première connexion (nombre de secondes depuis le 01/01/1970). Il y a raisonnablement peu de chances que deux utilisateurs utilisent la même machine au même moment (sauf cas où, par exemple, plusieurs dizaines de personnes accèdent simultanément au site derrière une passerelle unique). Cela suffit pour cet exemple simple.

Pour la suite du script, on dispose de l'identifiant de session dans la variable `$id_session`. On affiche alors successivement le formulaire pour les entrées, les plats puis les desserts. À chaque fois, on insère dans la table *Commande* le choix effectué précédemment, et on récapitule l'ensemble des choix en les affichant dans la page HTML. C'est toujours l'identifiant de session qui permet de faire le lien entre ces informations. Notez que la requête SQL qui récupère les choix de la session courante est une *jointure* qui fait appel à deux tables. Si vous ne connaissez pas SQL, les jointures sont présentées dans le chapitre 7, page 289. Le langage SQL dans son ensemble fait l'objet du chapitre 10. Les figures 2.7 et 2.8 montrent respectivement le formulaire après choix de l'entrée et du plat, et après le choix du dessert.

Dans ce script, nous devons intégrer des éléments d'un tableau associatif dans une chaîne de caractères. Quand il s'agit d'une variable simple, le fait que le nom de la variable soit préfixé par « $ » suffit pour que PHP substitue la valeur de la variable. C'est moins simple pour un tableau associatif. On ne peut pas écrire en effet :

```
echo "Ceci est un élément de tableau : $tab['code'] "; //Pas correct
```

Il existe deux manières correctes de résoudre le problème. Première solution, une concaténation :

```
echo "Ceci est un élément de tableau : ". $tab['code']; //Correct
```

Seconde solution : on encadre par des accolades l'élément du tableau pour qu'il n'y ait plus d'ambiguïté.

```
echo "Ceci est un élément de tableau : {$tab['code']} "; //Correct
```

Quand le dessert a été choisi, la session est terminée. Il faudrait alors demander confirmation ou annulation, et agir en conséquence dans la base de données. Ici nous nous contentons de détruire la commande, tâche accomplie !

Figure 2.7 — Après choix du plat et de l'entrée

Une commande au restaurant
http://localhost/exemples/ExSession.php
Faites votre commande au restaurant
Vous avez choisi : Crudités
Vous avez choisi : Turbot
Vous avez choisi : Paris-Brest
Nous avons noté votre commande. Merci !

Figure 2.8 — Le menu est choisi

2.5.3 Prise en charge des sessions dans PHP

PHP fournit un ensemble de fonctions pour faciliter la gestion des sessions. Ces fonctions permettent, pour l'essentiel :

1. d'engendrer automatiquement un identifiant de session, et de le récupérer à chaque nouvel accès ;

2. de stocker des informations associées à la session (par défaut le stockage a lieu dans un fichier temporaire) ;
3. de détruire toutes ces informations une fois la session terminée.

Dans la mesure où nous utilisons une base de données, une partie des fonctions de gestion de session, notamment celles qui consistent à conserver des informations sur les interactions passées de l'utilisateur, peuvent être avantageusement remplacées par des accès MySQL. L'intérêt est essentiellement de mieux protéger les accès aux donnés enregistrées dans le cadre de la session. Nous allons donc nous contenter des fonctions PHP essentielles à la gestion de session, données ci-dessous :

Fonction	Description
`session_start()`	Initialise les informations de session. Si aucune session n'existe, un identifiant est engendré et transmis dans un *cookie*. Si la session (connue par son identifiant) existe déjà, alors la fonction instancie toutes les variables qui lui sont liées. Cette fonction doit être appelée au début de tout script utilisant les sessions (il faut que l'instruction `Set-Cookie` puisse être placée dans l'en-tête HTTP).
`session_destroy()`	Détruit toutes les informations associées à une session.
`session_id()`	Renvoie l'identifiant de la session.

En général, un script PHP intégré dans une session débute avec `session_start()`, qui attribue ou récupère l'identifiant de la session (un *cookie* nommé `PHPSESSID`). On peut associer des variables à la session en les stockant dans le tableau `$_SESSION`. Une fois qu'une variable a été associée à une session, elle est automatiquement recréée et placée dans le tableau `$_SESSION` à chaque appel à `session_start()`. On peut la supprimer avec la fonction `unset()`, qui détruit une variable PHP. Enfin, quand la session est terminée, `session_destroy()` supprime toutes les variables associées (équivalent à un appel à `unset()` pour chaque variable).

Le script ci-dessous montre la gestion d'une session, équivalente à la précédente, avec les fonctions PHP.

Exemple 2.30 *exemples/SessionPHP.php* : *Gestion de session avec les fonctions PHP*

```
<?php
  // La fonction session_start fait tout le travail
  session_start();
?>
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Une commande au restaurant</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>
```

```
<h1>Faites votre commande au restaurant</h1>

<?php
require_once ("Connect.php");
require_once ("Connexion.php");
require_once ("ExecRequete.php");
require_once ("Normalisation.php");
require_once ("FormCommande.php");

// Connexion à la base
$connexion = Connexion (NOM, PASSE, BASE, SERVEUR);

// Normalisation des entrées HTTP
Normalisation();

// Si le type de choix n'est pas défini : on commence
// par proposer les entrées

if (!isSet($_POST['type_choix'])) {
  echo "Bonjour. Nous vous avons attribué la session "
    . session_id() . "<br/>";
  FormCommande ("Entrée", "SessionPHP.php", $connexion);
}
else {
  // Enregistrons le choix qui vient d'être fait
  // Il faudrait tester que id_choix est défini...
  $requete = "SELECT libelle FROM Carte "
    . "WHERE id_choix='$_POST[id_choix]'" ;
  $resultat = ExecRequete ($requete, $connexion);
  $choix = ObjetSuivant ($resultat);

  $_SESSION[$_POST['type_choix']] = $choix->libelle;

  // Affichage des choix déjà effectués
  if (isSet ($_SESSION['Entrée']))
    echo "Votre entrée : " . $_SESSION['Entrée'] . "<br/>";
  if (isSet ($_SESSION['Plat']))
    echo "Votre plat : " . $_SESSION['Plat'] . "<br/>";
  if (isSet ($_SESSION['Dessert']))
    echo "Votre dessert : " . $_SESSION['Dessert'] . "<br/>";

  // Affichage de la suite
  if ($_POST['type_choix'] == 'Entrée')
    FormCommande ("Plat", "SessionPHP.php", $connexion);
  else if ($_POST['type_choix'] == 'Plat')
    FormCommande ("Dessert", "SessionPHP.php", $connexion);
  else {
      echo "<p>Nous avons noté votre commande. Merci !<p/>";
      session_destroy ();
  }
}
```

```
?>
</body>
</html>
```

Les deux différences principales sont, d'une part, le recours à la fonction `session_start()` qui remplace la manipulation explicite des *cookies* (voir le script *ExSession.php*, page 103), et d'autre part l'utilisation du tableau `$_SESSION` à la place de la table *Commande*.

Ce tableau peut être vu comme une variable PHP persistante entre deux échanges client/serveur. Cette persistance est obtenue en stockant les valeurs du tableau dans un fichier temporaire, situé par exemple sous Unix dans le répertoire */tmp* (et configurable avec le paramètre `session_save_path` dans le fichier *php.ini*).

Ce mécanisme, valable pour la mise en place d'un système de gestion des sessions très simple, trouve rapidement ses limites. Si de très nombreuses informations doivent être associées à une session, il est préférable de les placer dans la base de données, en les référençant par l'identifiant de session (donné par la fonction `session_id()`). Une base de données permet de mieux structurer les informations. Elle persiste sur une très longue durée, contrairement à un fichier temporaire. D'autre part, elle est plus sûre puisque seuls les utilisateurs autorisés peuvent y accéder.

Les principes de gestion de session présentés ici seront repris de manière plus étendue dans le chapitre 7 pour développer des utilitaires robustes et associer la gestion de sessions à l'authentification des utilisateurs dans une base MySQL. Les fonctionnalités de PHP présentées précédemment nous suffiront puisque nous utilisons MySQL, mais vous pouvez consulter les autres fonctions PHP dans la documentation si vous pensez avoir à utiliser le mécanisme natif PHP. Il est possible en particulier d'éviter l'utilisation des *cookies* en demandant à PHP la réécriture de chaque URL dans une page pour y inclure l'identifiant de session. Comme expliqué au début de cette section, cette méthode reste cependant globalement insatisfaisante et peu sûre.

2.6 SQL DYNAMIQUE ET AFFICHAGE MULTI-PAGES

Dans la plupart des cas les requêtes SQL exécutées dans les scripts PHP sont fixées par le programmeur et ce dernier connaît le type du résultat (nombre d'attributs et noms des attributs). Il peut arriver que les ordres SQL soient « dynamiques », c'est-à-dire déterminés à l'exécution. C'est le cas par exemple quand on permet à l'utilisateur d'effectuer directement des requêtes SQL sur la base et d'afficher le résultat sous forme de table. On peut alors faire appel à des fonctions MySQL qui donnent des informations sur le type du résultat.

Voici une illustration de cette fonctionnalité avec, en guise de garniture, l'affichage « multi-pages » du résultat. Au lieu de donner en bloc, dans une seule page HTML, toutes les lignes du résultat de la requête, on affiche seulement un sous-groupe de taille fixée (ici, 10), et on donne la possibilité de passer d'un groupe à l'autre avec des ancres.

2.6.1 Affichage d'une requête dynamique

Commençons par écrire une fonction qui prend en argument le résultat d'une requête (tel qu'il est rendu par la fonction `mysql_query()`), la position de la première ligne à afficher, et le nombre de lignes à afficher.

Exemple 2.31 *exemples/AfficheResultat.php* : *Affichage partiel du résultat d'une requête SQL*

```
<?php
// Affichage partiel du résultat d'une requête

require ("UtilBD.php");

function AfficheResultat ($resultat, $position, $nbrLignes)
{

  // Affichage d'un tableau HTML, avec autant de colonnes
  // que d'attributs. On affiche $nbrLignes lignes,
  // à partir de la ligne indiquée par $position,

  echo "<table border='4'>";

  $compteurLignes = 1;
  $nbAttr = mysql_num_fields ($resultat);
  $noLigne=0;
  while ($tabAttr = mysql_fetch_row ($resultat)) {
    // Avant la première ligne, on affiche l'en-tête de la table
    if ($compteurLignes == 1) {
      echo "<tr>";
      // Affichage des noms d'attributs
      for ($i=0; $i < $nbAttr; $i++)
      echo "<th>" . mysql_field_name ($resultat, $i) . "</th>\n";
    }

    // Affichage de chaque ligne dans la fourchette [première,
       dernière]
    if ($compteurLignes >= $position
    and $compteurLignes <= $position + $nbrLignes -1) {
      $classe = "A" . (($noLigne++) % 2);
      echo "<tr class='$classe'>";
      for ($i=0; $i < $nbAttr; $i++) {
        if (empty($tabAttr[$i])) $tabAttr[$i] = "Champ vide";
        echo "<td>$tabAttr[$i]</td>";
      }
      echo "</tr>\n";
    }

    // Inutile de continuer si tout est affiché
    if ($compteurLignes++ >= $position + $nbrLignes - 1) break;
  }
```

```
  echo "</table>\n";
}
?>
```

La fonction `AfficheResultat()` utilise quelques nouvelles fonctionnalités de l'interface MySQL/PHP. Elles permettent d'obtenir la description du résultat, en plus du résultat lui-même.

1. `mysql_num_fields()` donne le nombre d'attributs dans le résultat ;
2. `mysql_field_name()` donne le nom de l'un des attributs ;
3. `mysql_fetch_row()` renvoie la ligne sous forme d'un tableau indicé, plus facile à manipuler que les tableaux associatifs ou les objets quand on doit exploiter le résultat de requêtes quelconques pour lesquelles on ne connaît pas, *a priori*, le type du résultat et donc le nom des attributs.

L'affichage comprend deux boucles. La première, classique, permet de parcourir toutes les lignes du résultat. Notez qu'ici on ne prend en compte que les lignes à présenter, à savoir celles dont la position est comprise entre `$position` et `$position+$nbrLignes-1`. La seconde boucle parcourt, pour une ligne donnée, tous les attributs.

```
echo "<tr class='A'" . (($noLigne++) % 2) . ">";
for ($i=0; $i < $nbAttr; $i++) {
  if (empty($tabAttr[$i])) $tabAttr[$i] = "Champ vide";
  echo "<td>$tabAttr[$i]</td>";
}
echo "</tr>\n";
```

On alterne la couleur de fond pour rendre la table plus lisible. Notre feuille de style, *films.css*, définit deux couleurs de fond pour les classes `A0` et `A1`.

```
tr.A0 {background-color:white}
tr.A1 {background-color:yellow}
```

On utilise alternativement les classes `A0` et `A1` pour la balise <**tr**>. On concatène pour cela le caractère '`A`' avec le résultat de l'expression `$l++ % 2`. La variable `$l++` est un entier qui, auto-incrémenté par l'opérateur '++', vaut successivement 0, 1, 2, 3, etc. En prenant cette valeur modulo 2 (l'opérateur '%'), on obtient l'alternance souhaitée de 0 et de 1.

2.6.2 Affichage multi-pages

Voyons maintenant comment réaliser l'affichage multi-pages, une technique très utile pour afficher de longues listes et utilisée, par exemple, par les moteurs de recherche.

Exemple 2.32 *exemples/ExecSQL.php* : *Affichage multi-pages du résultat d'une requête*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<head>
<title>Interrogation avec SQL</title>
<link rel='stylesheet' href="films.css" type="text/css" />
</head>
<body>

<h1>Interrogation avec SQL</h1>

<form method='post' action='ExecSQL.php'>
<textarea name='requete' cols='50' rows='3'><?php
if (isSet($_REQUEST['requete']))
echo $_REQUEST['requete'];
else
echo "SELECT * FROM FilmSimple";
?>
   </textarea>
<br />
<input name='submit' type='submit' value='Exécuter' />
</form>

<?php
require_once ("UtilBD.php");
require_once ("Normalisation.php");
require_once ("AfficheResultat.php");
define ("TAILLE_GROUPE", 10);

// Connexion à la base
$connexion = Connexion (NOM, PASSE, BASE, SERVEUR);

// Normalisation des entrées HTTP
Normalisation();

// La requête existe ? Il faut la traiter.
if (isSet($_REQUEST['requete'])) {
  $resultat = ExecRequete ($_REQUEST['requete'], $connexion);

  // On code la requête pour la placer dans une URL
  $requeteCodee = urlEncode($_REQUEST['requete']);

  // On vient de soumettre la requête dans le formulaire ? Dans
  // ce cas la
  // première ligne doit être affichée. Sinon on récupère la
     position courante
  if (isSet($_POST['submit'])) {
    $position = 1;
```

```
    }
    else {
      $position = $_GET['position'];
    }
    // Affichage des ancres pour les groupes qui suivent et/ou
    // précèdent
    if ($position > TAILLE_GROUPE) {
      // Il y a des lignes à voir avant
      $avant = $position - TAILLE_GROUPE;
      echo "<a href='ExecSQL.php?position=$avant&requete=
          $requeteCodee'>"
      ."Voir les ".TAILLE_GROUPE." lignes précédentes</a><br/>\n";
    }
    if ($position + TAILLE_GROUPE - 1 < mysql_num_rows($resultat))
        {
      // Il y a des lignes à voir après
      $apres = $position + TAILLE_GROUPE;
      echo "<a href='ExecSQL.php?position=$apres&requete=
          $requeteCodee'>"
      ."Voir les ".TAILLE_GROUPE." lignes suivantes</a><br/>\n";
    }

    // Affichage du résultat
    AfficheResultat ($resultat, $position, TAILLE_GROUPE);
}
?>
</body>
</html>
```

Le script comprend deux parties. Dans la première on présente un simple formulaire permettant de saisir une requête SQL (on réaffiche comme texte par défaut la requête saisie précédemment le cas échéant). La seconde partie, en PHP, est plus intéressante. Tout d'abord, on commence par récupérer la requête transmise par `post` ou `get` (on utilise donc le tableau `$_REQUEST` qui contient les deux, voir page 22), et on l'exécute. Notez qu'aucun traitement n'est appliqué à la requête car on suppose que l'utilisateur entre une syntaxe correcte, y compris l'échappement pour les « ' » dans les critères de sélection.

Ensuite, on regarde quelle est la partie du résultat à afficher. Si l'on vient du formulaire, la variable `$submit` est définie, et la position de départ est toujours 1. Sinon, la position est transmise dans l'URL (méthode `get`) et on la récupère.

On peut alors créer une ou deux ancres, selon le cas, pour accéder aux 10 lignes précédentes et/ou aux 10 lignes suivantes. Bien entendu, cela n'a pas de sens de proposer les lignes précédentes si l'on est en train d'afficher la première, ni d'afficher les 10 suivantes si l'on affiche la dernière. La fonction `mysql_num_rows()` donne la position de la dernière ligne. L'URL contient les deux paramètres indispensables au bon fonctionnement du script, à savoir la position et la requête (traitée avec `urlEncode()`). Remarquez qu'il serait possible dès le départ d'afficher une ancre pour chacun des groupes de lignes constituant le résultat de la requête (« les dix premiers », « les dix suivants », etc.).

Finalement, l'appel à la fonction `AfficheResultat()` avec les paramètres appropriés se charge de l'affichage (figure 2.9).

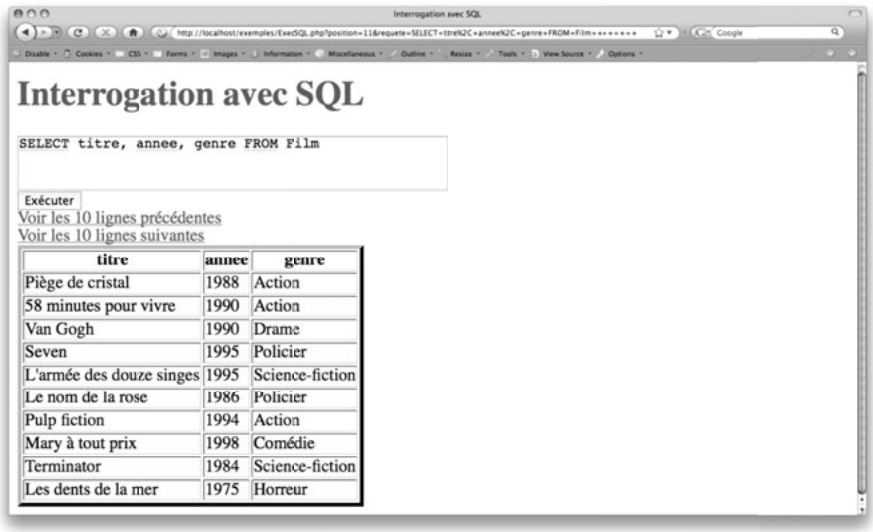

Interrogation avec SQL
SELECT titre, annee, genre FROM Film
Exécuter
Voir les 10 lignes précédentes
Voir les 10 lignes suivantes
titre annee genre
Piège de cristal 1988 Action
58 minutes pour vivre 1990 Action
Van Gogh 1990 Drame
Seven 1995 Policier
L'armée des douze singes 1995 Science-fiction
Le nom de la rose 1986 Policier
Pulp fiction 1994 Action
Mary à tout prix 1998 Comédie
Terminator 1984 Science-fiction
Les dents de la mer 1975 Horreur

Figure 2.9 — Le formulaire d'interrogation, avec affichage multi-pages

Cette technique « simule » une interactivité avec l'utilisateur par réaffichage d'un contenu modifié en fonction du contexte (ici la position courante), contenu lui-même obtenu par une opération (la saisie d'une requête) qui a pu s'effectuer longtemps auparavant. En d'autres termes, comme dans le cas des sessions, on établit une continuité de dialogue avec l'internaute en palliant les faiblesses de HTTP/HTML :

- l'absence d'interactivité d'une page HTML (sauf à recourir à des techniques sophistiquées comme JavaScript ou Flash) est compensée par des appels répétés au serveur ;
- HTTP ne gardant aucune mémoire des accès précédents, on prend soin de fournir les informations cruciales décrivant le contexte dans les messages (ici, la requête et la position courante) à chaque accès. Les URL incluses dans une page sont donc codées de manière à transmettre ces informations.

Ce script mérite quelques améliorations, omises pour en faciliter la lecture. Il faudrait effectuer des contrôles et prévoir des situations comme l'absence de résultat pour une requête. Par ailleurs, le choix de réexécuter systématiquement la requête n'est pas toujours le meilleur. Si elle est complexe à évaluer, cela pénalise le client (qui attend) et le serveur (qui travaille). D'autre part, si quelqu'un ajoute ou supprime en parallèle des lignes dans les tables concernées (voire supprime toutes les lignes) l'affichage sera décalé. Si ces problèmes se posent, une autre solution est d'exécuter la requête la première fois, de stocker le résultat dans une table ou un fichier temporaire, et de travailler ensuite sur ce dernier. Ces améliorations sont laissées au lecteur à titre d'exercice.

3

Programmation objet

Ce chapitre est entièrement consacré à la programmation objet avec PHP. D'un point de vue technique et conceptuel, son contenu est certainement l'un des plus avancés de ce livre, mais sa lecture n'est pas indispensable à la compréhension des chapitres qui suivent. Il est tout à fait possible de se contenter d'un premier survol consacré à *l'utilisation* de modules objet prêts à l'emploi, et de poursuivre la lecture avant d'y revenir éventuellement par la suite pour explorer la conception et l'implantation orientée-objet.

Comme l'ensemble du livre, ce chapitre est basé sur une approche concrète, avec comme souci constant de présenter les concepts à l'aide d'exemples réalistes et utilisables en pratique dans de véritables applications. Bien entendu la clarté recherchée impose certaines limitations sur les contrôles ou sur certaines fonctionnalités, mais l'une des caractéristiques de la programmation objet est de permettre des extensions qui ne remettent pas en cause le cœur de l'implantation, fournissant par là-même de bons sujets d'approfondissement. Rappelons que le site associé à ce livre propose un document énumérant des exercices d'application à partir des exemples donnés.

Par ailleurs, le chapitre peut se lire selon deux optiques : celle des « utilisateurs » qui exploitent des fonctionnalités orientées-objet, et celle des concepteurs et réalisateurs. Il semble indispensable de maîtriser la première optique puisque l'on trouve maintenant de très nombreuses fonctionnalités réalisées en suivant une approche orientée-objet, dont l'intégration, qui peut permettre d'économiser beaucoup de temps, suppose une connaissance des principes de base de cette approche. La seconde optique, celle du développeur, demande une conception de la programmation qui constitue un débouché naturel de celle basée sur des fonctions ou des modules, déjà explorée dans les chapitres précédents. On peut tout à fait se passer de la programmation objet pour réaliser une application, mais cette technique apporte inconstestablement un plus en termes de maîtrise de la complexité et de la taille du code, ainsi (mais c'est une question de goût) qu'un certain plaisir intellectuel à

produire des solutions simples et élégantes à des problèmes qui ne le sont pas toujours. Le contenu de ce chapitre est une tentative de vous convaincre sur ce point.

Depuis sa version 5, PHP est devenu un langage orienté-objet tout à fait respectable, même s'il n'atteint pas encore le niveau de complexité d'une référence comme le C++. La première section de ce chapitre est une présentation générale des concepts de la programmation orientée-objet, tels qu'ils sont proposés par PHP. Cette présentation est illustrée par une interface d'accès à un SGBD en général, et à MySQL en particulier. La syntaxe de la partie objet de PHP 5 est présentée successivement par les exemples, mais on peut également la trouver, sous une forme concise et structurée, dans le chapitre récapitulatif sur le langage (page 419). La programmation objet s'appuie sur un ensemble riche et souvent assez abstrait de concepts, ce qui impose probablement aux néophytes plusieurs lectures, en intercalant l'étude des exemples concrets qui suivent, pour bien les assimiler.

La suite du chapitre consiste, pour l'essentiel, en plusieurs exemples concrets de programmation objet visant à réaliser les fonctionnalités de base d'une application PHP/MySQL : outre l'accès à la base de données, on trouve donc la mise en forme des données avec des tableaux HTML, la production de formulaires, et enfin un « squelette » d'application, à la fois prêt à l'emploi et reconfigurable, permettant d'effectuer des opérations de mise à jour sur le contenu d'une table MySQL grâce à une interface HTML. Le niveau de difficulté va croissant, le dernier exemple exploitant de manière poussée la capacité de la programmation objet à réaliser des solutions « génériques », le moins dépendantes possibles d'un contexte particulier. À chaque fois, les deux optiques mentionnées précédemment sont successivement présentées :

- l'optique *utilisateur* : comment exploiter et faire appel aux fonctionnalités des objets ;
- l'optique *développeur* : comment elles sont conçues et réalisées.

Un des buts de la programmation objet est d'obtenir des modules fonctionnels (des « objets ») de conception et développement parfois très complexes, mais dont l'utilisation peut rester extrêmement simple. Tous les exemples décrits dans ce chapitre seront repris pour la réalisation de l'application décrite dans la partie suivante. Il suffira alors de bénéficier de la simplicité d'utilisation, en oubliant la complexité de la réalisation. Vous pourrez appliquer ce principe de réutilisation à vos propres développements, soit en y incluant les fonctionnalités décrites dans ce chapitre (et fournies sur le site), soit en récupérant les innombrables solutions fournies sur les sites de développement PHP (voir notamment le site *www.developpez.com*).

3.1 TOUR D'HORIZON DE LA PROGRAMMATION OBJET

Programmer, c'est spécifier des actions à exécuter au moyen d'un langage qui fournit des outils à la fois pour *concevoir* et pour *décrire* ces actions. On distingue classiquement, parmi ces outils, les *structures de données* qui permettent de représenter l'information à manipuler, et les *algorithmes* qui décrivent la séquence d'instructions

nécessaires pour effectuer l'opération souhaitée. Dans les chapitres précédents, les principales structures manipulées sont les variables et parfois des tableaux, et les algorithmes ont été implantés soit directement dans des scripts, soit sous forme de fonctions. Il y a donc, dans l'approche suivie jusqu'à présent, une séparation nette entre les *traitements* (les fonctions) et les *données* (variables, tableaux), considérées comme des informations transitoires échangées entre les fonctions.

3.1.1 Principes de la programmation objet

La programmation orientée-objet propose une intégration plus poussée des données et des traitements qui leur sont appliqués. Elle permet de masquer les informations qui ne servent qu'à une partie bien spécifique de l'application (par exemple la gestion des échanges avec la base de données) et de regrouper dans un module cohérent ces informations et les opérations qui portent sur elles. L'ensemble obtenu, données et traitement, constitue un *objet*, simplement défini comme un sous-système chargé de fournir des services au reste de l'application.

Les langages objets fournissent des outils très puissants pour la conception et la description des actions constituant une application. Concevoir une application objet, c'est d'abord imaginer un espace où des objets coopèrent en assumant chacun une tâche spécialisée. Cette approche permet de penser en termes de communications, d'interactions entre sous-systèmes, ce qui est souvent plus naturel que d'utiliser des outils conceptuels plus techniques comme les structures de données ou les fonctions.

Dans une perspective classique, non orientée-objet, on considère une application PHP comme un outil généraliste qui doit savoir tout faire, depuis la production de code HTML jusqu'à l'interrogation d'une base de données, en passant par les échanges avec des formulaires, la production de tableaux, etc. Cette approche présente des limites déjà soulignées pour la maîtrise des évolutions et de la maintenance quand le code atteint une certaine taille (quelques milliers de lignes). En introduisant des objets dans l'application, on obtient des « boîtes noires » dont le fonctionnement interne est inconnu du reste de l'application, mais capables de réaliser certaines tâches en fonction de quelques demandes très simples.

Prenons un cas concret correspondant aux objets que nous allons développer dans le cadre de ce chapitre. La figure 3.1 montre une application PHP classique (un moteur de recherche par exemple) constituée d'un formulaire pour saisir des critères, d'un accès à une base de données pour rechercher les informations satisfaisant les critères, et enfin d'un tableau pour afficher le résultat. Cette application s'appuie sur trois objets :

1. un objet `Formulaire` chargé de produire la description HTML du formulaire ;
2. un objet `BD` qui communique avec la base de données ;
3. un objet `Tableau` qui effectue la mise en forme du résultat en HTML.

Chaque objet est doté d'un *état* constitué des données – ou *propriétés* – qui lui sont nécessaires pour l'accomplissement de ses tâches et d'un *comportement* constitué de

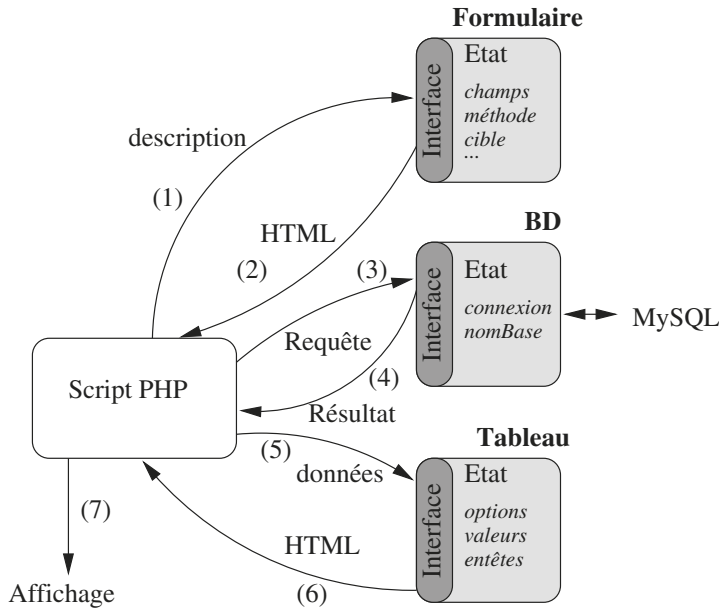


Figure 3.1 — Application avec objets.

l'ensemble des opérations – ou *méthodes* – qu'on peut appliquer à cet état. Dans le cas de l'objet `BD` l'état est par exemple l'identifiant de connexion et le nom de la base courante, et le comportement est constitué de méthodes de connexion, d'exécution de requêtes, et de parcours du résultat. Pour l'application, il reste à contrôler ces objets en spécifiant les différentes opérations nécessaires (numérotées de 1 à 7 sur la figure) pour arriver au résultat souhaité.

Méthodes et encapsulation

Les propriétés et méthodes constituant l'état et le comportement d'un objet peuvent être *privées* ou *publiques*. Il est fortement recommandé de cacher les propriétés d'un objet en les rendant privées, pour qu'on ne puisse y accéder que par l'intermédiaire des méthodes de cet objet. Les propriétés d'un objet « fichier » comme son nom, son emplacement, son état (ouvert, fermé), ne regardent pas l'utilisateur de l'objet. Cette dissimulation, désignée par le terme *d'encapsulation*, offre de nombreux avantages dont, par exemple, la possibilité de revoir complètement la description interne d'un objet sans remettre en cause les applications qui l'utilisent, ces dernières n'en voyant que les méthodes. Le principe est donc de ne publier qu'un sous-ensemble des méthodes donnant accès sous la forme la plus simple et la plus puissante possible aux fonctionnalités proposées par l'objet. L'application doit juste connaître les méthodes publiques permettant de demander à lun des objets de déclencher telle ou telle opération.

Voyons maintenant un exemple concret de programmation objet, consacré à l'interrogation de la table *FilmSimple* et à l'affichage de toutes ses lignes. Nous avons déjà vu que quand on accède à MySQL et que l'on demande une ligne d'une table sous la forme d'un objet, les fonctions d'interfaçage PHP/MySQL créent automatiquement cet objet, sans restriction d'accès sur les propriétés. Si `$film` est un objet, alors on accède librement aux attributs `$film->titre` et `$film->annee` qui permettent respectivement d'obtenir le titre et l'année.

Pour nos propres objets, nous ferons toujours en sorte que les propriétés soient privées et ne puissent être manipulées que par l'intermédiaire de l'interface constituée de méthodes. Le tableau 3.1 donne la liste des méthodes publiques de la classe `MySQL`.

Tableau 3.1 — Les méthodes publiques de la classe `MySQL`

Méthode	Description
__construct(*login, motDePasse, base, serveur*)	Constructeur d'objets.
execRequete(*requête*)	Exécute une requête et renvoie un identifiant de résultat.
objetSuivant(*résultat*)	Renvoie la ligne courante sous forme d'objet et avance le curseur d'une ligne.
ligneSuivante(*résultat*)	Renvoie la ligne courante sous forme de tableau associatif et avance le curseur d'une ligne.
tableauSuivant(*résultat*)	Renvoie la ligne courante sous forme de tableau indicé et avance le curseur d'une ligne.
__destruct()	Se déconnecte du serveur de base de données.

Exemple 3.1 *exemples/AppliClasseMySQL.php* : *Application d'un objet.*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Connexion à MySQL</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>Interrogation de la table FilmSimple</h1>

<?php
require_once ("Connect.php");
require_once ("MySQL.php");

try{
  $bd = new MySQL (NOM, PASSE, BASE, SERVEUR);

  $resultat = $bd->execRequete ("SELECT * FROM FilmSimple");
```

```
    while ($film = $bd->objetSuivant ($resultat))
     echo "<b>$film->titre</b>, paru en $film->annee, réalisé "
       . "par  $film->prenom_realisateur $film->nom_realisateur.
                                                    <br/>\n";
}
catch (Exception $exc) {
  echo "<b>Erreur rencontrée:</b> " . $exc->getMessage() . "\n";
}
?>
</body>
</html>
```

Ce premier exemple montre trois objets d'origines différentes à l'œuvre :

1. l'objet `$film` nous est déjà familier : il représente une ligne du résultat d'une requête et comprend autant d'attributs – publics – que de colonnes dans ce résultat ;
2. l'objet `$bd` est destiné à interagir avec la base de données ; il est créé grâce à une « fabrique » à objets – une *classe* – nommée `MySQL` ;
3. enfin, l'objet `$exc` est une *exception* PHP, créée quand une erreur survient quelque part dans le code ; cet objet contient les informations nécessaires à la gestion de l'erreur.

On peut déjà remarquer la concision et la clarté du code, obtenues grâce au masquage de nombreuses informations – par exemple l'identifiant de connexion à la base – ou instructions inutiles. Ce code correspond en fait strictement aux actions nécessaires à la logique de la fonctionnalité implantée : accès à une base, exécution d'une requête, parcours et affichage du résultat avec gestion des erreurs potentielles.

Une bonne partie du code masqué est placé dans l'objet `$bd` qui fournit les services de connexion, d'interrogation et d'exploration du résultat d'une requête. Nous allons étudier la manière dont cet objet est créé, avant de passer à la gestion des exceptions.

3.1.2 Objets et classes

Comment faire pour définir soi-même ses propres objets ? On utilise des *classes*, constructeurs décrivant les objets. Une classe est un « moule » qui permet de créer à la demande des objets conformes à la description de la classe. Il y a la même distinction entre une classe et ses objets, qu'entre le type `string` et l'ensemble des chaînes de caractères, ou entre le schéma d'une table et les lignes de cette table. On appelle *instances* d'une classe les objets conformes à sa description.

Une classe définit non seulement les propriétés communes à tous ses objets, mais également leur *comportement* constitué, comme nous l'avons vu, de l'ensemble des méthodes qu'on peut leur appliquer. Un objet ne doit pas être seulement vu comme un ensemble de propriétés, mais aussi – et surtout – comme un (petit) système fournissant des services au script qui l'utilise. Un objet fichier, par exemple, devrait

fournir des services comme `ouvrir` (le fichier), `fermer` (le fichier), `lire` (une ligne, ou un mot), `écrire`, etc.

La classe `MySQL` regroupe les fonctionnalités de connexion et d'accès à MySQL. Toutes les spécificités liées à MySQL sont dissimulées dans la classe et invisibles de l'extérieur, ce qui permet de généraliser facilement le code à d'autres SGBD de manière transparente pour le reste de l'application, comme nous le verrons plus loin.

Une classe en PHP se définit (comme en C++ ou en Java) par un bloc `class { ... }` qui contient à la fois les propriétés de la classe et les méthodes, désignées par le mot-clé `function`. Propriétés et méthodes peuvent être qualifiées par les mots-clés `public`, `private` ou `protected`, ce dernier correspondant à une variante de `private` sur laquelle nous reviendrons au moment de discuter de l'héritage et de la spécialisation.

Voici le code complet de la classe `MySQL`. Cette implantation assez simplifiée suffira pour un premier exemple. Une version de cette classe, étendue et améliorée, est proposée avec le code du site *Films*. Notez enfin que PHP propose de manière native (depuis la version 5.1) une implantation orientée-objet assez semblable dans son principe à celle que je présente ici, PDO (*Persistent Data Objects*). Il est évidemment préférable d'utiliser PDO dans un site professionnel, plutôt que mes classes, à visée essentiellement didactique, même si elles fonctionnent très bien.

Exemple 3.2 *exemples/MySQL.php* : *La classe* `MySQL`.

```
<?php
// Une classe de gestion des accès à une base MySQL (version
// simplifiée)
class MySQL
{
  // ——— Partie privée : les propriétés
  private $connexion, $nomBase;

  // Constructeur de la classe
  function __construct ($login, $motDePasse, $base, $serveur)
  {
    // On conserve le nom de la base
    $this->nomBase = $base;

    // Connexion au serveur MySQL
    if (!$this->connexion = @mysql_pconnect ($serveur, $login,
        $motDePasse))
    throw new Exception ("Erreur de connexion au serveur.");

    // Connexion à la base
    if (!@mysql_select_db ($this->nomBase, $this->connexion))
    throw new Exception ("Erreur de connexion à la base.");
  }
  // Fin du constructeur
```

```
  // ——— Partie publique ———————————————————

  // Méthode d'exécution d'une requête
  public function execRequete ($requete)
  {
    $resultat = @mysql_query ($requete, $this->connexion);

    if (!$resultat)
    throw new Exception
    ("Problème dans l'exécution de la requête : $requete. "
    . mysql_error($this->connexion));

    return $resultat;
  }

  // Accès à la ligne suivante, sous forme d'objet
  public function objetSuivant ($resultat)
  { return  mysql_fetch_object ($resultat);    }
  // Accès à la ligne suivante, sous forme de tableau associatif
  public function ligneSuivante ($resultat)
  { return  mysql_fetch_assoc ($resultat);  }
  // Accès à la ligne suivante, sous forme de tableau indicé
  public function tableauSuivant ($resultat)
  { return  mysql_fetch_row ($resultat);  }

  // Échappement des apostrophes et autres préparations à
  // l'insertion
  public function prepareChaine($chaine)
  { return mysql_real_escape_string($chaine);  }

  // Destructeur de la classe: on se déconnecte
  function __destruct ()
  { @mysql_close ($this->connexion);    }
  // Fin de la classe
}
```

La classe comprend quatre parties : les propriétés, le constructeur, les méthodes privées (ou protégées) et publiques, enfin le destructeur.

> **REMARQUE –** Vous noterez que le fichier ne se termine pas par la balise fermante PHP. Cela évite les problèmes dus aux caractères parasites qui pourraient suivre cette balise et empêcher la production d'en-têtes HTTP. L'absence de cette balise ne pose pas de problème pour l'interpréteur PHP.

Les *propriétés* décrivent l'état d'un objet instance de la classe. Nous avons ici l'identifiant de connexion à MySQL et le nom de la base courante. Ces deux variables sont accessibles dans toutes les méthodes, publiques ou privées, avec la syntaxe `$this->connexion` et `$this->nomBase`. De plus, leur valeur persiste tout au long de la durée de vie d'un objet, contrairement aux variables locales d'une

fonction classique. Pour des raisons exposées précédemment, les propriétés sont *privées*. Elles peuvent donc être utilisées dans les méthodes de la classe (préfixées par `$this->`), mais restent inaccessibles pour une application manipulant un objet. Toute interaction passe nécessairement par les méthodes publiques.

Le constructeur est une méthode (optionnelle) spéciale, ayant soit le nom `__construct` (avec deux caractères '_'), soit le même nom que la classe. Si un constructeur est défini, il est exécuté au moment où un nouvel objet est créé, ce qui permet donc d'une part d'affecter une valeur initiale, si besoin est, aux propriétés de l'objet, d'autre part d'effectuer les tâches initiales de cet objet. Ici, on se connecte au serveur MySQL à la base choisie. Les instructions `throw` correspondent à des « lancers » d'exception quand une erreur est rencontrée : nous y revenons plus loin. Notez l'utilisation de `@` préfixant les fonctions MySQL, pour éviter l'affichage incontrôlé de messages d'erreur si un problème survient (en effet, l'opérateur `@` peut s'appliquer à n'importe quelle expression PHP pour annuler les messages d'erreur).

Après exécution du constructeur, si aucune erreur n'est rencontrée, les propriétés `$connexion` et `$nomBase` sont correctement initialisées et prêtes à être utilisées par les méthodes. Pour construire un objet, on utilise l'opérateur `new` suivi d'un appel au constructeur (ou simplement du nom de la classe si on n'a pas défini de constructeur). Voici par exemple la création d'une connexion à MySQL.

```
require_once ("Connect.php");
require_once ("MySQL.php");

$bd = new MySQL (NOM, PASSE, BASE, SERVEUR);
```

Les constantes `NOM`, `PASSE`, `BASE` et `SERVEUR` sont définies dans le fichier *Connect.php*, ce qui permet de les modifier très facilement pour toute l'application. La variable `$bd` est maintenant un objet sur lequel on va pouvoir appliquer toutes les méthodes de la classe `MySQL`.

Les méthodes publiques correspondent aux fonctionnalités de base de MySQL. Notez qu'on ne peut pas conserver l'identifiant du résultat d'une requête comme variable interne au même titre que `$connexion` car, pour un même objet instance de la classe, on peut envisager d'exécuter simultanément plusieurs requêtes. Il existe donc potentiellement plusieurs identifiants de résultats valides à un moment donné. L plus simple pour les gérer est de les échanger avec le script appelant pour désigner la requête concernée.

```
// On récupère un identifiant de résultat
$resultat = $bd->execRequete ("SELECT * FROM FilmSimple");

// On s'en sert pour désigner le résultat qu'on veut parcourir
while ($film = $bd->objetSuivant ($resultat)) {...}
```

La dernière méthode, `__destruct`, est le *destructeur* que l'on trouve dans des langages orientés-objets plus évolués comme C++. La notion de destructeur est introduite en PHP 5. Notons que la présence d'un destructeur n'est pas indispensable, et souvent de peu d'utilité en PHP où les ressources sont libérées automatiquement en fin de script. Ici, on ferme la connexion à MySQL.

3.1.3 Les exceptions

Voyons maintenant le mécanisme de « lancer » d'exception. Il répond au problème suivant : un programme (ou un script) un tant soit peu complexe peut être vu comme un arbre composé d'appels successifs à des fonctions qui effectuent des tâches de plus en plus spécialisées au fur et à mesure que l'on s'enfonce dans la hiérarchie. On est souvent embarassé pour traiter les erreurs dans un tel programme car la manière dont l'erreur doit être gérée dépend du contexte – le programme appelant – parfois éloigné de l'endroit où l'erreur est survenue. La situation classique est illustrée sur un cas simple dans la partie gauche de la figure 3.2. Le programme A appelle une fonction B qui appelle elle-même une fonction C, où l'erreur est rencontrée. On peut alors considérer tout un ensemble de solutions parmi les deux extrêmes suivants :

1. on reporte l'erreur au moment où on la rencontre, soit dans la fonction C pour notre exemple ;
2. on remonte l'erreur, de C vers A, par des passages de paramètres successifs vers le programme appelant.

La première solution, simple, a l'inconvénient de ne pas permettre l'adaptation à un contexte particulier. L'impossibilité de se connecter à une base de données par exemple peut être, selon le programme, soit une erreur fatale entraînant l'arrêt total, soit une erreur bénigne qui peut être compensée par d'autres actions (par exemple le recours à une base de secours). Effectuer la décision d'arrêt ou de poursuite au niveau de la procédure de connexion n'est donc pas satisfaisant.

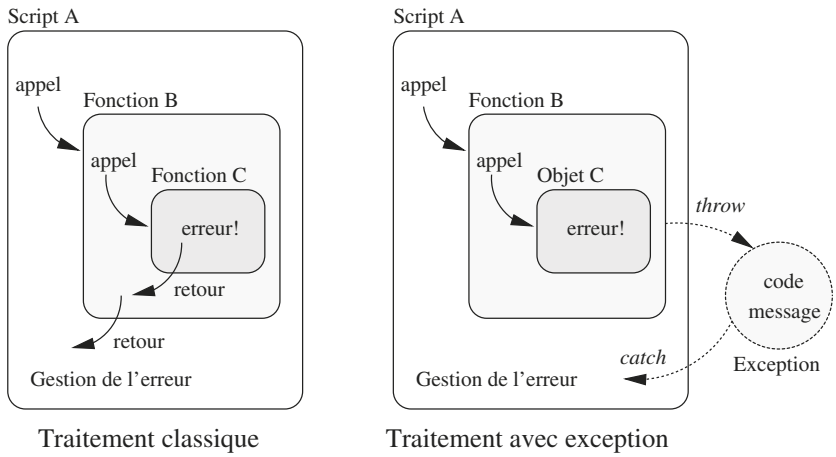


Figure 3.2 — Gestion des exceptions.

La seconde solution est conceptuellement correcte puisqu'elle permet de décider à quel niveau de la hiérarchie des appels on va traiter l'erreur rencontrée. Elle est cependant, dans un cadre classique de programmation par fonctions, pénible à mettre en œuvre à cause de la nécessité d'une part de gérer la « remontée » des erreurs avec des paramètres ou des valeurs de retour, et d'autre part de devoir détecter sans cesse, après un appel à une fonction, la présence d'une erreur.

Le mécanisme de « lancer » d'exception facilite considérablement cette remontée d'erreur. Il est maintenant répandu dans de nombreux langages (comme C++, Java ainsi que d'autres non spécifiquement objet comme le PL/SQL d'Oracle). La partie droite de la figure 3.2 illustre le principe :

- quand une erreur est rencontrée, on « lance » (*throw* en anglais) une exception, placée dans un espace réservé du programme ;
- à chaque niveau de la hiérarchie des appels, on peut « attraper » (*catch* en anglais) l'exception levée auparavant par une fonction ou une méthode appelée.

Le fait de placer les exceptions dans un espace séparé évite d'avoir à les inclure dans les paramètres des fonctions ou méthodes. De plus, une exception est un objet qui fournit plusieurs informations sur le contexte de l'erreur : un message, un code d'erreur (optionnel), le fichier et le numéro de la ligne de l'instruction PHP qui a déclenché l'erreur. Ces informations sont respectivement obtenues par les méthodes `getMessage()`, `getCode()`, `getFile()` et `getLine()` de la classe prédéfinie `Exception`.

Voici un exemple de gestion d'exception dans la classe `MySQL`. Au niveau du constructeur, on lance l'exception si la procédure de connexion a échoué :

```
function __construct ($login, $motDePasse, $base, $serveur)
{
  ...

  // Connexion au serveur MySQL
  if (!$this->connexion = @mysql_pconnect ($serveur, $login,
      $motDePasse))
    throw new Exception ("Erreur de connexion au serveur.");
  ...
}
```

Quand l'instruction `throw` est déclenchée, l'exécution de la méthode est interrompue et l'exception est mise en attente. Tout programme appelant la méthode (ou appelant une fonction appelant la méthode, et ainsi de suite) peut « attraper » cette exception et la traiter. Il faut pour cela utiliser la construction suivante :

```
try {
 // Espace d'interception des exceptions lancées
}
catch (Exception $e)
{
 // Traitement de l'exception lancée
}
```

Le bloc `try` définit la partie du script qui va « attraper » toute exception lancée par un appel à une méthode effectuée dans le bloc. Quand une exception est attrapée dans le bloc `try`, le flux d'exécution se redirige immédiatement vers le bloc `catch` qui récupère l'exception en la plaçant dans une variable (ici `$e`) et peut alors la traiter. Voici comment nous avons géré les exceptions de la classe `MySQL` dans le script `ApplClasseMySQL.php`, page 119.

```
try{
  $bd = new MySQL (NOM, PASSE, BASE, SERVEUR);
  ...
}
catch (Exception $exc)
{
  echo "<b>Erreur rencontrée:</b> " . $exc->getMessage() . "\n";
}
```

Il est évidemment possible de décider au cas par cas de la gestion d'une exception. On peut se contenter de l'afficher, comme dans l'exemple ci-dessus, ou bien envoyer un e-mail à l'administrateur et afficher un message neutre et poli à l'utilisateur si l'on ne souhaite pas exhiber une faiblesse du site. Le chapitre 5 reviendra en détail sur la politique de gestion des erreurs.

3.1.4 Spécialisation : classes et sous-classes

Un concept essentiel en programmation objet est la *spécialisation* : il désigne la possibilité de créer des sous-classes définissant des objets plus spécialisés que ceux de la classe-mère. Si on considère par exemple une classe `Fichier` avec des méthodes d'ouverture, de fermeture, d'affichage et de lecture/écriture dans le fichier, on peut ensuite spécialiser le concept de fichier avec des sous-classes `FichierTexte`, `FichierImage`, `FichierBinaire`, `FichierRépertoire`, etc. En PHP, on dit qu'une sous-classe *étend* sa classe parente. Chaque objet instance de la sous-classe est *aussi* une instance de la super-classe et peut être traité comme tel si c'est nécessaire. Dans notre exemple, chaque instance de `FichierTexte` est aussi instance de `Fichier`. Voici un squelette des définitions possibles de la classe `Fichier` et de deux de ses sous-classes :

```
class Fichier {
  // Propriétés
  private $nom;

  // Constructeur
  public __construct ($nomFichier) { }

  // Une méthode
  public copier ($destination) {...}
}

// Sous-classe des fichiers texte
class FichierTexte extends Fichier
{
  // Propriétés
  private $contenu;

  // Ajout d'une méthode
  public afficher($nom_imprimante) {...}
}
```

```
// Sous-classe des fichiers répertoire
class Répertoire extends Fichier
{
  // Propriétés
  private $liste_fichiers;

  // Surcharge de la méthode copier()
  public copier ($destination) {/* Définition propre aux
      répertoires*/}
}
```

Cet exemple très partiel est essentiellement destiné à illustrer les principaux concepts liés à la spécialisation : *héritage* des propriétés ou méthodes de la super-classe, *ajout* de propriétés ou de méthodes dans les sous-classes, *surcharge* de méthodes. Voyons cela dans l'ordre.

Héritage

La notion d'héritage découle directement de celle de spécialisation. Dans la mesure où un objet instance d'une sous-classe est aussi une instance de la super-classe, la sous-classe doit disposer – ou « hériter » – de toutes les propriétés et méthodes de cette dernière. Par exemple les fichiers textes étant des fichiers particuliers, toutes les descripions ou opérations valables pour les fichiers au sens générique du terme le sont également pour un fichier texte. Un objet instance de `FichierTexte` dispose donc, sans qu'il soit besoin de rien préciser ou redéfinir, de la propriété `nom` et de la méthode `copier()`.

L'héritage permet de considérer l'ensemble des instances d'une classe C, *ou de n'importe laquelle de ses classes descendantes* comme des objets uniformes dotés du même comportement, celui de la classe C. Dans notre exemple, les fichiers texte ou les répertoires (et aussi les fichiers images, les fichiers exécutables ou toute autre instance d'une sous-classe de `Fichier`) ont un nom et disposent de la méthode `copier()`, ce qui peut servir par exemple à effectuer une sauvegarde en appliquant systématiquement cette méthode sans se soucier du type précis de fichier manipulé. En d'autres termes, on « factorise » au niveau de la super-classe le comportement commun à toutes les instances de ses descendantes, facilitant ainsi les traitements qui n'ont pas besoin d'effectuer de distinction entre les différentes spécialisations.

Ajout de nouvelles propriétés ou méthodes

Dans certaines circonstances, on souhaite au contraire considérer un objet comme une instance d'une classe spécialisée dotée de caractéristiques particulières. La classe `FichierTexte` illustre cet aspect : les objets instances de cette classe ont, *en plus des propriétés et méthodes de la classe* `Fichier`, des propriétés et méthodes propres :

- la propriété `contenu` permet de stocker sous forme de chaîne de caractères le contenu du fichier ;
- la méthode `afficher` rend possible le rendu de ce contenu.

Ces caractéristiques sont propres à ce type de fichier : on n'imagine pas de gérer le contenu d'un fichier exécutable ou de l'afficher. L'ajout de propriétés ou de méthodes

qui raffinent la description des objets de la super-classe est un aspect inséparable de la spécialisation.

Surcharge

Enfin la *surcharge* est le mécanisme qui consiste à enrichir, voire à remplacer complètement, un comportement défini au niveau de la super-classe par un autre, adapté aux caractéristiques de la classe spécialisée.

> **REMARQUE –** Attention, la documentation PHP utilise le terme « surcharge » (*overloading*) dans un sens différent de celui consacré en programmation objet. La notion de surcharge présentée ici est conforme avec celle classique, rencontrée en C++ ou en Java.

Dans notre exemple la méthode `copier()` de la sous-classe `Répertoire` doit être implantée différemment de la méthode codée au niveau de la classe `Fichier`, car, outre la copie du fichier-répertoire lui-même, on doit également copier l'ensemble des fichiers contenus *dans* le répertoire. Ce comportement de la méthode `copier()` est tout à fait spécifique à ce type de fichier et nécessite toute la « surcharge » – la redéfinition – de la méthode héritée. Voici, très simplifié, l'essentiel des instructions que l'on pourrait trouver dans cette surcharge :

```
class Répertoire {
  // Propriétés
  private $liste_fichiers;

  // Une méthode
  public copier ($destination)
  {
    // On commence par copier le répertoire lui-même
    // en appelant la méthode de la super-classe
    parent::copier($destination);

    // Puis on copie tous les fichiers contenus
    foreach ($this->liste_fichier as $fichier)
      $fichier->copier($destination);
  }
```

Cette méthode se décompose clairement en deux parties : l'une consistant à effectuer une copie standard, telle qu'elle est définie au niveau de la classe parent, l'autre répercutant la demande de copie sur l'ensemble des fichiers contenus dans le répertoire et référencés par la propriété ajoutéee `liste_fichiers`.

Pour appliquer la copie standard, on doit appeler le code défini au niveau de la classe parente. On utilise pour cela la construction `parent::copier`. Cette pratique est d'usage dans tous les cas, fréquents, où la surcharge consiste à *enrichir* le comportement défini au niveau de la classe générique, ce qui implique de conserver ce comportement tout en lui ajoutant de nouvelles instructions. Ces nouvelles instructions consistent ici à parcourir l'ensemble des fichiers contenus en leur appliquant à leur tour la méthode *copier()*.

```
foreach ($this->liste_fichiers as $fichier)
  $fichier->copier($destination);
```

À chaque étape de la boucle, la variable `$fichier` référence un des fichiers contenus dans le répertoire, et on demande à cet objet de se copier vers la destination. Il s'agit d'un excellent exemple du processus d'abstraction consistant à voir selon les circonstances ces fichiers comme des objets « génériques » (instance de la classe `Fichier`) ou comme des objets spécialisées, instances des sous-classes de `Fichier`. Il faut imaginer ici, pour se limiter à notre exemple, que les fichiers contenus dans un répertoire peuvent être soit des fichiers texte, soit eux-mêmes des répertoires contenant d'autres fichiers. En les considérant uniformément, dans la boucle, comme des instances de `Fichier` dotés d'une méthode de copie, on s'évite le souci d'avoir à distinguer les différents types d'actions à effectuer en fonction du type précis de fichier manipulé, et on laisse à l'objet lui-même le soin de déterminer la méthode à appliquer.

Ce type de programmation peut sembler subtil quand on y est confronté les premières fois, mais il s'acquiert d'autant plus vite qu'on est convaincu du gain apporté par un raisonnement en termes génériques sans avoir à se soucier à chaque instant des détails d'implantation. La simplicité du code obtenu une fois qu'on a résolu le problème de la conception et de la modélisation d'une application objet vient largement compenser l'effort initial à fournir.

3.1.5 Spécialisation et classes abstraites : la classe BD

Voyons un exemple complet qui nous permettra également d'introduire un dernier concept. La suite du chapitre consistera à approfondir, par la conception et l'implantation de plusieurs classes, tout ce qui est résumé ici.

Notre exemple consiste ici à définir, en recourant à la spécialisation objet, un ensemble de classes définissant de manière uniforme les accès à une base de données relationnelle, quelle qu'elle soit. Nous allons prendre comme cibles MySQL, PostgreSQL et ORACLE, avec comme objectif la possibilité de définir des applications qui utilisent indifféremment l'un ou l'autre système, et ce de manière totalement transparente. Le site décrit dans la seconde partie de l'ouvrage s'appuie sur ces classes pour rendre le code compatible avec tout système relationnel.

> **REMARQUE –** Le code proposé ici fonctionne correctement, mais il est surtout conçu comme une illustration des concepts orientés-objet. Comme signalé précédemment, l'interface PDO de PHP offre une solution normalisée et plus complète. Reportez-vous page 238 pour une introduction à PDO.

En termes de spécialisation, il n'y a aucune raison de dire qu'une classe définissant les interactions avec PostgreSQL *hérite* de celle accédant à MySQL, ou l'inverse. La bonne question à se poser est toujours « un objet instance de la classe spécialisée est-il aussi un objet de la classe générique ? ». La réponse est clairement non puisqu'un objet accédant à MySQL n'est pas un objet accédant à PostgreSQL et vice-versa. En revanche tous deux sont des exemples d'un concept commun, celui d'objet accédant

à une base de données. Ce concept commun n'a pas d'instanciation : il n'existe pas d'objet qui fournisse ce comportement indépendamment d'un choix concret d'une base de données spécifique.

Quand on a besoin de définir un comportement commun à un ensemble d'objets sans que ce comportement puisse être directement instancié, on utilise la notion de *classe abstraite* qui permet de factoriser la description des méthodes fournies par tous les objets instances des classes spécialisées, à charge pour ces classes de définir l'implantation appropriée de chacune de ces méthodes. Dans notre cas, il s'agit de définir toutes les méthodes (au sens précis de : noms, liste des paramètres en entrée et en sortie, rôle de la méthode) communes à tous les objets, quel que soit le SGBD auquel ils permettent d'accéder. Il nous faut au minimum :

1. une méthode de connexion ;
2. une méthode d'exécution des requêtes ;
3. une ou plusieurs méthodes pour récupérer le résultat ;
4. une méthode pour traiter les apostrophes ou autres caractères gênants dans les chaînes à insérer dans les requêtes (la technique d'échappement pouvant varier d'un SGBD à un autre) ;
5. une gestion des erreurs.

Au moment de la définition de ces méthodes, il faut s'assurer qu'elles correspondent à des fonctionnalités qui peuvent être fournies par *tous* les SGBD. Il faut également réfléchir soigneusement aux paramètres d'entrée et de sortie nécessaires à chaque méthode (on désigne souvent par *signature* cette spécification des paramètres). En d'autres termes, on doit définir de manière générique, c'est-à-dire sans se soucier des détails d'implantation, l'interface d'accès à un SGBD en évitant de se laisser influencer, à ce stade, par les particularités de l'un d'entre eux. Voici la classe abstraite `BD`.

Exemple 3.3 *exemples/BD.php* : *La classe abstraite* `BD`

```
<?php
// Classe abstraite définissant une interface générique d'accès
// à une base de données. Version simplifiée: une définition
// plus complète est donnée avec le site Films

abstract class BD
{
  // ——— Partie privée : les propriétés
  protected $connexion, $nom_base;

  // Constructeur de la classe
  function __construct ($login, $mot_de_passe, $base, $serveur)
  {
    // On conserve le nom de la base
    $this->nom_base = $base;
```

```
    // Connexion au serveur par appel à une méthode privée
    $this->connexion = $this->connect($login, $mot_de_passe,
    $base, $serveur);

    // Lancé d'exception en cas d'erreur
    if ($this->connexion == 0)
    throw new Exception ("Erreur de connexion au SGBD");

    // Fin du constructeur
  }

  // Méthodes privées
  abstract protected function connect ($login,
            $mot_de_passe, $base, $serveur);
  abstract protected function exec ($requete);

  // Méthodes publiques

  // Méthode d'exécution d'une requête
  public function execRequete ($requete)
  {
    if (!$resultat = $this->exec ($requete))
    throw new Exception
    ("Problème dans l'exécution de la requête : $requete.<br/> "
    . $this->messageSGBD());

    return $resultat;
  }

  // Méthodes abstraites
  // Accès à la ligne suivante, sous forme d'objet
  abstract public function objetSuivant ($resultat);
  // Accès à la ligne suivante, sous forme de tableau associatif
  abstract public function ligneSuivante ($resultat);
  // Accès à la ligne suivante, sous forme de tableau indicé
  abstract public function tableauSuivant ($resultat);

  // Echappement des apostrophes et autres préparations à
  // l'insertion
  abstract public function prepareChaine($chaine);

  // Retour du message d'erreur
  abstract public function messageSGBD ();
  // Fin de la classe
}
```

La définition d'une classe abstraite est préfixée par le mot-clé `abstract`, de même que toutes les méthodes de la classe pour lesquelles seule la signature est donnée. Toute classe PHP comprenant au moins une méthode abstraite doit elle-même être déclarée comme abstraite.

REMARQUE – La notion de classe abstraite existe depuis PHP 5, de même que celle *d'interface*, concept assez proche mais encore un peu plus générique, que nous ne présentons pas ici.

Dans la classe `BD` toutes les méthodes sont abstraites, à l'exception du constructeur et de la méthode `execRequete()` qui exécute une requête. Ces deux méthodes montrent comment répartir les tâches entre classe abstraite et classe dérivée :

1. tout ce qui est *commun* à l'ensemble des SGBD doit être factorisé au niveau de la classe abstraite : ici il s'agit de la réaction à adopter si une connexion ou l'exécution d'une requête échoue (on a choisi en l'occurrence de lever une exception) ;
2. tout ce qui est *spécifique* à un système particulier doit être dévolu à une méthode abstraite qui devra être implantée au niveau de chaque classe dérivée (ici on a donc deux méthodes abstraites `connect()` et `exec()` destinées à fournir respectivement le code de connexion et d'exécution de requêtes propres à chaque système).

La mention `protected` introduite ici est une variante de `private`. Elle signifie que la méthode ou la propriété est invisible de tout script appelant (comme si elle était privée) mais accessible en revanche à toute sous-classe qui peut donc la surcharger. Toute propriété ou méthode déclarée `private` n'est accessible que dans la classe qui la définit. La méthode `exec()` par exemple doit être déclarée `protected` pour pouvoir être redéfinie au niveau des sous-classes de `BD`.

Il est impossible d'instancier un objet d'une classe abstraite. Celle-c n'est d'une certaine manière qu'une spécification contraignant l'implantation des classes dérivées. Pour être instanciables, ces classes dérivées doivent impérativement fournir une implantation de toutes les méthodes abstraites. Voici la classe `BDMySQL` (à comparer avec la classe `MySQL`, page 121).

Exemple 3.4 *exemples/BDMySQL.php* : *La classe dérivée* `BDMySQL`

```
<?php
// Sous-classe de la classe abstraite BD, implantant l'accès à
// MySQL

require_once("BD.php");

class BDMySQL extends BD
{
  // Pas de propriétés: elles sont héritées de la classe BD
  // Pas de constructeur: lui aussi est hérité

  // Méthode connect: connexion à MySQL
  protected function connect ($login, $mot_de_passe, $base,
      $serveur)
  {
    // Connexion au serveur MySQL
```

```
    if (!$this->connexion = @mysql_pconnect ($serveur, $login,
        $mot_de_passe))
    return 0;

    // Connexion à la base
    if (!@mysql_select_db ($this->nom_base, $this->connexion))
    return 0;

    return $this->connexion;
  }

  // Méthode d'exécution d'une requête.
  protected function exec ($requete)
  { return @mysql_query ($requete, $this->connexion); }

  // Partie publique: implantation des méthodes abstraites
  // Accès à la ligne suivante, sous forme d'objet
  public function objetSuivant ($resultat)
  { return  mysql_fetch_object ($resultat);    }
  // Accès à la ligne suivante, sous forme de tableau associatif
  public function ligneSuivante ($resultat)
  { return  mysql_fetch_assoc ($resultat); }
  // Accès à la ligne suivante, sous forme de tableau indicé
  public function tableauSuivant ($resultat)
  { return  mysql_fetch_row ($resultat); }

  // Echappement des apostrophes et autres préparation à
  // l'insertion
  public function prepareChaine($chaine)
  { return mysql_real_escape_string($chaine); }

  // Retour du message d'erreur
  public function messageSGBD ()
  { return mysql_error($this->connexion);}

  // Méthode ajoutée: renvoie le schéma d'une table
  public function schemaTable($nom_table)
  {
    // Recherche de la liste des attributs de la table
    $liste_attr = @mysql_list_fields($this->nom_base,
    $nom_table, $this->connexion);

    if (!$liste_attr) throw new Exception ("Pb d'analyse de
        $nom_table");

    // Recherche des attributs et stockage dans le tableau
    for ($i = 0; $i < mysql_num_fields($liste_attr); $i++) {
      $nom =  mysql_field_name($liste_attr, $i);
      $schema[$nom]['longueur'] = mysql_field_len($liste_attr, $i
          );
      $schema[$nom]['type'] = mysql_field_type($liste_attr, $i);
      $schema[$nom]['cle_primaire'] =
```

```
        substr_count(mysql_field_flags($liste_attr, $i), "
            primary_key");
        $schema[$nom]['not_null'] =
        substr_count(mysql_field_flags($liste_attr, $i), "not_null");
      }
      return $schema;
    }

    // Destructeur de la classe: on se déconnecte
    function __destruct ()
    {if ($this->connexion)  @mysql_close ($this->connexion);     }
    // Fin de la classe
}
?>
```

On peut noter que la redéfinition du constructeur est inutile puisqu'il est déjà fourni au niveau de la classe parente. En revanche, il faut en définir la partie spécifique, soit les méthodes `connect()` et `exec()`. Au moment où on effectuera une instanciation d'un objet de la classe `BDMySQL`, l'exécution se déroulera comme suit :

- le constructeur défini dans la classe `BD` sera appelé, puisqu'il est hérité, et non surchargé ;
- ce constructeur appelle à son tour la méthode `connect()` qui, elle, est définie au niveau de la classe `BDMySQL`.

Le constructeur lèvera une exception si la méthode `connect()` échoue. On a bien l'interaction souhaitée entre le code générique de la classe parente et le code spécifique de la classe dérivée. Le même mécanisme s'applique à l'exécution de requêtes, avec la méthode générique `execRequete()` appelant la méthode spécifique `exec()` (ainsi, éventuellement, que la méthode `messageSGBD()`), et levant une exception si nécessaire en fonction du retour de cette dernière. Cela étant, une classe publique de la super-classe peut toujours être surchargée. Si on souhaite par exemple lever deux exceptions différentes, une pour l'erreur de connexion au serveur et l'autre pour l'erreur d'accès à une base, on peut redéfinir un constructeur pour la classe `BDMySQL` comme suit :

```
function __construct ($login, $mot_de_dasse, $base, $serveur)
{
  // On conserve le nom de la base
  $this->nom_base = $base;

  // Connexion au serveur MySQL
  if (!$this->connexion =
        @mysql_pconnect ($serveur, $login, $mot_de_dasse))
    throw new Exception ("Erreur de connexion au serveur.");

  // Connexion à la base
  if (!@mysql_select_db ($this->nom_base, $this->connexion))
```

```
        throw new Exception ("Erreur de connexion à la base.");
}
```

Attention : quand une méthode est surchargée (donc redéfinie dans une classe dérivée), la méthode de la classe parente n'est plus appelée. La surcharge est donc bien un remplacement de la méthode héritée. C'est valable également pour le constructeur : la définition d'un constructeur pour la classe `BDMySQL` implique que le constructeur de la super-classe `BD` ne sera plus appelé au moment de l'instanciation d'un objet `BDMySQL`. Il est cependant possible de faire l'appel explicitement grâce à la syntaxe `parent::BD()` : voir l'exemple de la classe `Répertoire`, page 128.

Toutes les autres méthodes abstraites sont ensuite implantées par un simple appel à la fonction correspondante de l'interface de programmation (API) MySQL. Il est bien entendu possible d'étendre la puissance de la classe dérivée en lui ajoutant d'autres fonctionnalités de MySQL. Ces méthodes seraient alors spécifiques aux instances de la classe `BDMySQL` et ne pourraient donc pas être appelées dans une application souhaitant pouvoir accéder à des SGBD différents et se fondant sur l'interface définie dans la super-classe `BD`.

Regardons de plus près la méthode `schemaTable`. Si tout se passe bien, elle renvoieun tableau associatif à deux dimensions décrivant pour chaque attribut (première dimension du tableau) les options de création de la table passée en paramètre (seconde dimension). Il s'agit à peu de choses près des informations du `CREATE TABLE` : longueur et type d'un attribut donné, et booléen indiquant si cet attribut fait partie de la clé primaire identifiant une ligne de la table.

Cette fonction renvoie une valeur dont la taille peut être importante. Cette valeur, initialement stockée dans une variable *locale* de la méthode, `schemaTable`, doit ensuite être *copiée* vers une variable du script appelant. Ce code est correct mais on peut se poser la question de l'impact négatif sur les performances en cas d'appels intensifs à cette méthode pour des tables contenant beaucoup d'attributs. L'utilisation d'un passage par référence peut alors s'envisager (voir la discussion page 61). On aurait le simple changement :

```
function schemaTable($nom_table, &$schema) {
 // Comme avant
 ...
}
```

et la fonction alimenterait directement la variable du script appelant, dont on obtient ici une référence.

La méthode `schemaTable()` est une méthode ajoutée (elle sera utilisée pour une autre classe, page 167). La déclarer sous forme de méthode abstraite au niveau de la classe `BD` enrichirait la spécification des interactions, mais imposerait l'implantation de cette méthode dans toutes les sous-classes.

Il reste à définir autant de sous-classes que de SGBD, soit ORACLE, ou PostgreSQL, ou encore SQLite, un moteur SQL directement intégré à PHP depuis la version 5, etc. La classe ci-dessous correspond à PostgreSQL.

Exemple 3.5 *exemples/BDPostgreSQL.php* : *La classe dérivée* `BDPostgreSQL`

```
<?php
// Sous-classe de la classe abstraite BD, implantant l'accès à
// PostgreSQL

require_once("BD.php");

class BDPostgreSQL extends BD
{
  // Pas de propriétés: elles sont héritées de la classe BD
  // Pas de constructeur: lui aussi est hérité

  // Méthode connect: connexion à PostgreSQL
  protected function connect ($login, $mot_de_passe, $base,
      $serveur)
  {
    // Quelques ajustements PostgreSQL...
    $login = strToLower($login);  $base = strToLower($base);
    if ($serveur == 'localhost') $serveur ="";
    // Création de la chaîne de connexion
    $chaineC = "user=$login dbname=$base password=$mot_de_passe
        host=$serveur";
    // Connexion au serveur et à la base
    return $this->connexion = pg_connect ($chaineC);
  }

  // Méthode d'exécution d'une requête
  protected function exec ($requete)
  { return @pg_exec ($this->connexion, $requete);  }

  // ---- Partie publique ----------------------------
  // Accès à la ligne suivante, sous forme d'objet
  function objetSuivant ($resultat)
  { return  pg_fetch_object ($resultat);    }
  // Accès à la ligne suivante, sous forme de tableau associatif
  function ligneSuivante ($resultat)
  { return  pg_fetch_assoc ($resultat);  }
  // Accès à la ligne suivante, sous forme de tableau indicé
  function tableauSuivant ($resultat)
  { return  pg_fetch_row ($resultat);  }

  // Echappement des apostrophes et autres préparations à
  // l'insertion
  public function prepareChaine($chaine)
  { return addSlashes($chaine);  }

  // Retour du message d'erreur
  public function messageSGBD ()
  { return pg_last_error($this->connexion);}

  // Destructeur de la classe: on se déconnecte
```

```
    function __destruct ()
    { @pg_close ($this->connexion);     }
    // Fin de la classe
}
?>
```

On retrouve la même structure que pour `BDMySQL`, avec l'appel aux fonctions correspondantes de PostgreSQL, et la prise en compte de quelques spécificités. Caractéristique (assez désagréable...) de l'interface PHP/PostgreSQL : tous les identificateurs (noms de tables, d'attributs, de base, etc.) sont systématiquement traduits en minuscules, ce qui impose quelques conversions avec la fonction PHP `strToLower()` (voir la méthode *connect()* ci-dessus). De plus, pour la connexion au serveur `localhost`, PostgreSQL demande que le nom du serveur soit la chaîne vide. Ces particularités peuvent être prises en compte au moment de l'implantation des méthodes abstraites.

On peut maintenant considérer qu'un objet instance de la classe `BDMySQL` ou un objet instance de la classe `BDPostgreSQL` sont tous deux conformes au comportement décrit dans la super-classe commune, `BD`. On peut donc les utiliser *exactement* de la même manière si on se limite au comportement commun défini dans cette super-classe. Le script suivant montre un code qui, hormis le choix initial de la classe à instancier, fonctionne aussi bien pour accéder à MySQL que pour accéder à PostgreSQL (ou SQLite, ou ORACLE, ou tout autre système pour lequel on définira une sous-classe de `BD`).

Exemple 3.6 *exemples/AppliClasseBD.php* : *Accès générique à un SGBD.*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Application de la classe BD</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<h1>Application de la classe BD</h1>

<?php
require_once ("Connect.php");

// La sous-classe pour MySQL
require_once ("BDMySQL.class.php");
// La sous-classe pour PostgreSQL
require_once ("BDPostgreSQL.class.php");
// La sous-classe pour SQLite
require_once ("BDSQLite.class.php");
```

```
try{
  if (isSet($_GET['postgresql']))
    $bd = new BDPostgreSQL (NOM, PASSE, BASE, SERVEUR);
  else    if (isSet($_GET['sqlite']))
    $bd = new BDSQLite (NOM, PASSE, BASE, SERVEUR);
  else
    $bd = new BDMySQL (NOM, PASSE, BASE, SERVEUR);

  $resultat = $bd->execRequete ("SELECT * FROM FilmSimple");

  while ($film = $bd->objetSuivant ($resultat))
    echo "<b>$film->titre </b>, paru en $film->annee, "
      . "réalisé par $film->prenom_realisateur $film->
          nom_realisateur<br/>";
}
catch (Exception $exc)
{
  echo "<b>Erreur rencontrée:</b> " . $exc->getMessage() . "\n";
}
?>
</body>
</html>
```

Si on passe une variable `postgresql` en mode `GET`, c'est à PostgreSQL qu'on se connecte, sinon c'est à MySQL, ou à SqLite, etc. Dans une application importante, détaillée dans la seconde partie de ce livre, on peut instancier initialement un objet en choisissant le SGBD à utiliser, et le passer ensuite en paramètre aux fonctions ou objets qui en ont besoin. Ceux-ci n'ont alors plus à se soucier de savoir à quel système ils accèdent, tant qu'ils se conforment au comportement de la classe générique.

> **REMARQUE –** Écrire une application multi-plateformes demande cependant quelques précautions supplémentaires. Il existe de nombreuses différences mineures entre les différents SGBD qui peuvent contrarier, et parfois compliquer, la production d'un code complètement compatible. La première précaution à prendre (nécessaire, mais par forcément suffisante...) est de respecter strictement la norme SQL du côté SGBD. Il faut ensuite étudier soigneusement les interfaces entre PHP et le SGBD pour détecter les points susceptibles de poser problème. Le fait que l'interface de PostgreSQL traduise tous les identificateurs en minuscules est par exemple une source d'incompatibilité à prendre en compte dès la conception, en n'utilisant que des identificateurs déjà en minuscules. Nous revenons en détail page 233 sur le problème de la portabilité multi-SGBD.

3.1.6 Résumé

Ce premier tour d'horizon a permis de voir l'essentiel des principes de la programmation objet. Si c'est votre premier aperçu de cette technique, il est probable que vous trouviez tout cela compliqué et inutilement abstrait. À l'usage, la cohérence de cette approche apparaît, ainsi que ses avantages, notamment en terme sde simplification de la programmation et de la maintenance. Il n'est pas obligatoire de programmer

en objet pour réaliser des applications robustes, et on peut envisager de ne pas maîtriser l'ensemble de la panoplie des concepts et techniques. La programmation PHP s'oriente cependant de plus en plus vers l'utilisation et la réutilisation d'objets prêts à l'emploi. La compréhension de ce mode de production du logiciel semble donc s'imposer.

Les exemples qui vont suivre permettent d'approfondir cette première présentation et de présenter quelques nouveautés qui sont brièvement résumées ci-dessous pour compléter cette première section.

Constantes. Il est possible en PHP 5 de définir des constantes locales à une classe. L'usage est principalement d'initialiser des valeurs par défaut utilisables par la classe et ses sous-classes.

Propriétés et méthodes statiques. Les méthodes ou propriétés vues jusqu'à présent étaient toujours considérées dans le cadre d'un objet de la classe. Chaque objet dispose d'une valeur propre pour chaque propriété, et les méthodes appliquées à un objet s'appliquent à ces valeurs. Les propriétés et méthodes statiques sont au contraire rattachées à la classe, pas à chacune de ses instances. Il en existe donc une unique copie par classe, utilisable, par exemple, pour compter le nombre d'objets instanciés à un moment donné.

Interfaces. Une interface, comme son nom l'indique, est la spécification d'une liste de fonctions avec leur nom et leur mode d'appel. Une classe abstraite propose le même type de spécification, implicitement destinée à s'appliquer aux instances de la classe. La notion d'interface est un peu plus générale dans la mesure où elle est définie indépendamment de toute classe, donc de toute instance. Une classe peut alors *implanter* une ou plusieurs interfaces. L'utilisation des interfaces permet de pallier en partie l'absence de concepts comme l'héritage multiple. Il s'agit cependant de techniques avancées qui dépassent le cadre de ce livre et ne seront donc pas détaillées.

Identité d'un objet. Un objet, c'est une identité et une valeur. Deux objets sont dits *identiques* s'ils ont la même identité, et *égaux* s'ils ont la même valeur. L'égalité se teste avec l'opérateur classique `==`, alors que l'identité se teste avec l'opérateur PHP5 `===` [1].

Important : quand on passe un objet en paramètre à une fonction, c'est son identité (ou sa référence, en terminologie PHP) qui est transmise, pas sa valeur. De même l'affectation `$a = $b;`, où `b` est un objet, fait de `a` une référence vers `b` (voir page 61). Les objets constituent donc une exception au principe de passage des paramètres par valeur en usage dans tous les autres cas. Concrètement, cela signifie que toute modification effectuée dans la fonction appelée agit directement sur l'objet, pas sur sa copie. Cette règle ne vaut que depuis la version 5, puisque PHP 4 (et versions antérieures) appliquaientt la règle du passage par valeur.

1. Ceux qui confondraient déjà l'opérateur d'affectation `=` et l'opérateur de comparaison `==` apprécieront !

Classes cibles L'opérateur `::` permet dans certains cas d'indiquer explicitement la classe dans laquelle chercher une définition. La syntaxe est `NomClasse::définition`, où `définition` est soit une constante de la classe `NomClasse`, soit une propriété ou une méthode statique. Deux mot-clés réservés peuvent remplacer `NomClasse:` `parent` et `self` qui désignent respectivement la classe parent et la classe courante. Quand une méthode est surchargée, ils peuvent indiquer quelle version de la méthode on souhaite appeler (voir par exemple page 128).

Le chapitre 11 complète cette rapide présentation et donne l'ensemble de la syntaxe objet de PHP. En ce qui concerne la modélisation des applications objet, rappelons que tout ce qui précède est repris, en PHP, d'autre langages orientés-objet, notamment du C++ et, dans une moindre mesure de Java. Pour aller plus loin dans l'approfondissement de la programmation objet, vous pouvez recourir à un ouvrage généraliste consacré au C++, à Java, ou à la conception objet en général.

3.2 LA CLASSE TABLEAU

La classe présentée dans cette section montre comment concevoir et réaliser un utilitaire de production de tableaux HTML qui évite d'avoir à multiplier sans cesse, au sein du code PHP, des balises <**tr**>, <**td**>, etc. Un tel utilitaire prend place dans une stratégie générale de séparation du code HTML et du code PHP sur laquelle nous reviendrons au chapitre 5.

La première chose à faire quand on projette la création d'une nouvelle classe, c'est de bien identifier les caractéristiques des objets instances de cette classe, leur représentation, les contraintes portant sur cette représentation, et enfin les méthodes publiques qu'ils vont fournir. À terme, ce qui nous intéresse, c'est la manière dont on va pouvoir communiquer avec un objet.

3.2.1 Conception

Le but est de produire des tableaux HTML. Il faut pour cela les construire dans l'objet, en attendant de pouvoir afficher par la suite le code HTML correspondant. Pour commencer, il faut se faire une idée précise de ce qui constitue un tableau et des options de présentation dont on veut disposer. On cherche le meilleur rapport possible entre la simplicité de l'interface des tableaux, et la puissance des fonctionnalités. On peut commencer par identifier les besoins les plus courants en analysant quelques cas représentatifs de ce que l'on veut obtenir, puis spécifier les données et traitements nécessaires à la satisfaction de ces besoins.

Les tableaux sont très utilisés en présentation de données statistiques. Prenons le cas d'une base d'information sur la fréquentation des films (aou « box office » pour faire court), classée selon divers critères comme les villes et la semaine d'exploitation, et voyons les différentes possibilités de représentation par tableau. La

première possibilité est simplement de mettre chaque donnée en colonne, comme ci-dessous.

Tableau 3.2 — Tableau A.

Film	Semaine	Ville	Nb entrées
Matrix	1	Paris	12000
Matrix	2	Paris	15000
Matrix	3	Paris	11000
Spiderman	1	Paris	8000
Spiderman	2	Paris	9000
Spiderman	3	Paris	9500
Matrix	1	Caen	200
Matrix	2	Caen	2100
Matrix	3	Caen	1900
Spiderman	1	Caen	1500
Spiderman	2	Caen	1600
Spiderman	3	Caen	1200

C'est la représentation qu'on obtient classiquement avec un SGBD relationnel comme MySQL. Elle est assez peu appropriée à la visualisation des propriétés du jeu de données (comme l'évolution du nombre d'entrées, ou les proportions entre les différents films). Voici une seconde possibilité qui montre, sur Paris, et par film, le nombre d'entrées au cours des différentes semaines.

Tableau 3.3 — Tableau B.

Box office	Semaine 1	Semaine 2	Semaine 3
Matrix	12000	15000	11000
Spiderman	8000	9000	9500

On s'est ici limité à deux dimensions, mais des artifices permettent de présenter des tableaux de dimension supérieure à 2. Voici par exemple une variante du tableau précédent, montrant les mêmes données sur Paris *et* sur Caen, ce qui donne un tableau à trois dimensions.

Tableau 3.4 — Tableau C.

Box office	Film	Semaine 1	Semaine 2	Semaine 3
Paris	Matrix	12000	15000	11000
	Spiderman	8000	9000	9500
Caen	Matrix	2000	2100	1900
	Spiderman	1500	1600	1200

Bien entendu on pourrait présenter les entrées dans un ordre différent, comme dans l'exemple ci-dessous.

Tableau 3.5 — Tableau D.

Box office	Semaine 1		Semaine 2		Semaine 3	
	Paris	Caen	Paris	Caen	Paris	Caen
Matrix	12000	2000	15000	2100	11000	1900
Spiderman	8000	1500	9000	1600	9500	1200

Une possibilité encore :

Tableau 3.6 — Tableau E.

Paris			
	Semaine 1	**Semaine 2**	**Semaine 3**
Matrix	12000	15000	11000
Spiderman	8000	9000	9500
Caen			
	Semaine 1	**Semaine 2**	**Semaine 3**
Matrix	200	2100	1900
Spiderman	1500	1600	1200

Tous ces exemples donnent dans un premier temps un échantillon des possibilités de présentation en clarifiant les caractéristiques des données qui nous intéressent. Ces deux aspects, présentation et données, sont en partie indépendants puisqu'à partir du même box office, on a réussi à obtenir plusieurs tableaux très différents.

L'étape suivante consiste à décrire dces tableaux de manière plus abstraite. Pour les données, nous pouvons distinguer les *dimensions*, qui servent au classement, et les *mesures* qui expriment la valeur constatée pour une combinaison donnée de dimensions [2]. Dans les exemples ci-dessus, les dimensions sont les films, les villes, les semaines, et la seule mesure est le nombre d'entrées. Autrement dit, le nombre d'entrées est fonction d'un film, d'une ville, et d'une semaine.

Veut-on gérer plusieurs mesures, c'est-à-dire présenter plusieurs valeurs dans une même cellule du tableau ? On va répondre « non » pour simplifier. D'une manière générale on a donc une fonction M qui prend en paramètres des dimensions $d_1, d_2, \ldots, d_p$ et renvoie une mesure m. On peut gérer cette information grâce à un tableau PHP multi-dimensionnel `$M[`d_1`][`d_2`]...[`d_p`]`. À ce stade il faut se demander si cela correspond, de manière suffisamment générale pour couvrir largement les besoins, aux données que nous voudrons manipuler. Répondons « oui » et passons à la présentation du tableau.

Un peu de réflexion suffit à se convaincre que si l'on souhaite couvrir les possibilités A, B, C, D et E ci-dessus, l'utilisation de la classe deviendra assez difficile pour l'utilisateur (ainsi bien sûr que la réalisation du code, mais cela importe moins puisqu'en principe on ne fera l'effort une fois et on n'y reviendra plus). Le cas du

2. Cette modélisation reprend assez largement la notation, le vocabulaire et les principes en usage dans les entrepôts de données, supports privilégiés de ce type de tableaux statistiques.

tableau E, assez éloigné des autres, sera ignoré. Voici, pour un tableau avec deux dimensions d_1 et d_2, la représentation adoptée.

CSG	$e[d_2, c_2^1]$	$e[d_2, c_2^2]$	...	$e[d_3, c_2^q]$
$e[d_1, c_1^1]$	$M[c_1^1, c_2^1]$	$M[c_1^1, c_2^2]$	...	$M[c_1^1, c_2^q]$
$e[d_1, c_1^2]$	$M[c_1^2, c_2^1]$	$M[c_1^2, c_2^2]$	...	$M[c_1^2, c_2^q]$
...	...	...	...	...
$e[d_1, c_1^p]$	$M[c_1^p, c_2^1]$	$M[c_1^p, c_2^2]$	...	$M[c_1^p, c_2^q]$

Tableau 3.7 — Les méthodes publiques de la classe `tableau`

Méthode	Description
Tableau(`tabAttrs`)	Constructeur de tableaux en fonction d'une dimension et d'une liste de paramètres de présentation.
ajoutValeur(`ligne, colonne, valeur`)	Définit la `valeur` du tableau dans une cellule donnée par les paramètres `ligne` et `colonne`.
ajoutEntete(`dimension, cle, texte`)	Définit l'en-tête pour la dimension `dimension` et la clé `cle`.
TableauHTML()	Produit la représentation HTML du tableau.
ajoutAttributsTable(`tabAttrs`)	Ajouts de paramètres de présentation pour la balise <**table**>.
setCouleurPaire(`couleur`)	Couleur de fond pour les lignes paires.
setCouleurImpaire(`couleur`)	Couleur de fond pour les lignes impaires.
setAfficheEntete(`dimension, couleur`)	Indique si l'on souhaite ou non afficher l'en-tête pour la dimension.
setCoinSuperieurGauche(`texte`)	Texte à placer dans le coin supérieur gauche.

Les éléments apparaissant dans cette présentation sont :

- Le libellé du coin supérieur gauche CSG (dans le tableau C par exemple c'est « Box office ») ;
- les clés de la dimension 1, notées c_1^i, pour chaque ligne i, avec $1 \leqslant i \leqslant p$ (dans le tableau B ce sont les titres de films ; dans le tableau C les villes) ;
- les clés de la dimension 2, notées c_2^j, pour chaque ligne j, avec $1 \leqslant j \leqslant q$ (dans notre exemple il s'agit de `'Semaine'` suivi du numéro de la semaine) ;
- les en-têtes de la dimension d_k, notés $e[d_k, c_k^i]$, avec $k = 1$ ou $k = 2$;
- enfin $M[i, j]$ désigne la valeur de la mesure pour la position (i, j) du tableau.

Une fois cet effort de modélisation effectué, tout le reste devient facile. Les informations précédentes doivent pouvoir être manipulées par l'intermédiaire de l'interface de la classe `Tableau` et donc être stockées comme propriétés des objets de la classe `Tableau`. Par ailleurs, elles doivent être accessibles en entrée ou en sortie par l'intermédiaire d'un ensemble de méthodes publiques.

Ce modèle de tableau capture les exemples A et B. En l'étendant à trois dimensions, on obtient également les présentations C et D. En revanche il ne convient pas

au tableau E : il faut savoir renoncer aux cas qui rendent beaucoup plus complexes les manipulations sans que cela soit justifié par le gain en puissance.

Dans ce qui suit, nous donnons des exemples d'utilisation, ainsi que l'implantation de la classe, en nous limitant au cas à deux dimensions. La gestion d'un nombre de dimensions quelconque est partiellement réalisée dans le code fourni sur le site, et partiellement laissée au lecteur (le polycopié d'exercices fournit des suggestions complémentaires).

3.2.2 Utilisation

La table 3.7 donne la liste des méthodes publiques de la classe `Tableau`. On trouve bien entendu le constructeur de la classe, qui prend en paramètres la dimension du tableau et des attributs HTML à placer dans la balise <**table**>. Les trois méthodes suivantes sont les plus importantes. Elles définissent respectivement l'ajout d'une valeur dans une cellule (le tableau M des mesures), la description des en-têtes (le tableau *e*) et enfin la sortie de la chaîne de caractères contenant la représentation HTML du tableau.

Les autres méthodes publiques sont moins essentielles. Elles permettent de régler l'apparence du tableau en affectant certaines valeurs à des paramètres internes à la classe utilisés ensuite au moment de la génération de la chaîne HTML.

Voyons maintenant comment on utilise cette classe dans une petite application de test qui extrait des données de MySQL et les affiche sous forme de tableau HTML. Le script SQL suivant permet de créer la table *BoxOffice* (les exemples contiennent un autre script, *InsBoxOffice.sql*, pour insérer un échantillon de données dans cette table).

Exemple 3.7 *exemples/BoxOffice.sql* : *Création de la table BoxOffice.*

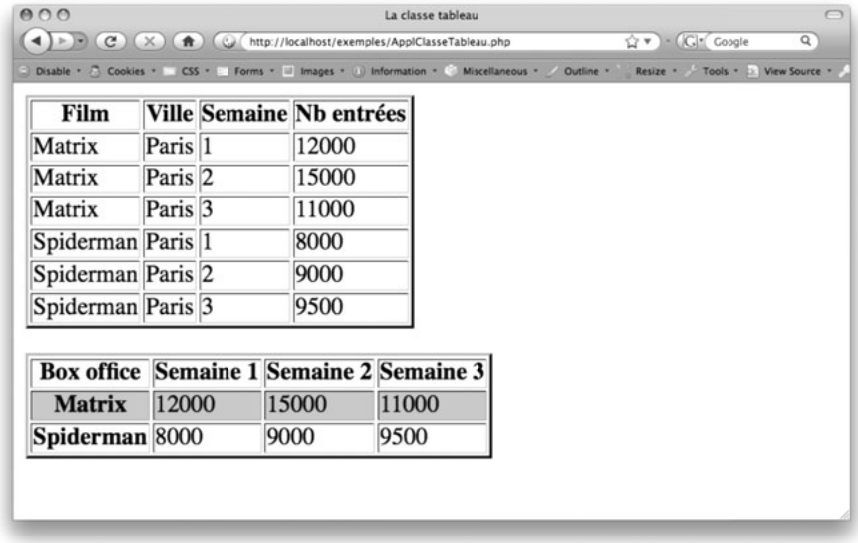
```
# Création d'une table pour box office simplifié

CREATE TABLE BoxOffice
(titre VARCHAR(60) NOT NULL,
 semaine INTEGER NOT NULL,
 ville   VARCHAR(60) NOT NULL,
 nb_entrees INTEGER NOT NULL,
 PRIMARY KEY (titre, semaine, ville)
);
```

Le script *AppIClasseTableau.php*, ci-dessous, instancie deux objets de la classe `Tableau`, correspondant aux présentations A et B données précédemment. Ces deux objets sont alimentés à partir des lignes issues d'une même requête, ce qui montre concrètement comment on peut facilement choisir une présentation particulière en partant des mêmes données. Notez qu'il n'y a pratiquement plus une seule balise HTML apparaissant dans ce script. La figure 3.3 donne le résultat.

Exemple 3.8 *exemples/AppliClasseTableau.php* : *Application de la classe* `Tableau`.

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>La classe tableau</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<?php
require_once ("Connect.php");
require_once ("BDMySQL.php");
require_once ("Tableau.php");

try{
  // Connexion à la base de données
  $bd = new BDMySQL (NOM, PASSE, BASE, SERVEUR);

  // Création du premier tableau
  $tableauA = new Tableau(2, array("border"=>2));
  $tableauA->setAfficheEntete(1, FALSE);

  // Création du second tableau
  $tableauB = new Tableau(2, array("border"=>2));
  $tableauB->setCoinSuperieurGauche("Box office");
  $tableauB->setCouleurImpaire("silver");

  $i=0;
  // Recherche des films parisiens
  $resultat = $bd->execRequete ("SELECT * FROM BoxOffice WHERE
      ville='Paris'");
  while ($bo = $bd->objetSuivant ($resultat)) {
      // Premier tableau: présentation standard, en colonnes
      $i++;
      $tableauA->ajoutValeur($i, "Film", $bo->titre);
      $tableauA->ajoutValeur($i, "Ville", $bo->ville);
      $tableauA->ajoutValeur($i, "Semaine", $bo->semaine);
      $tableauA->ajoutValeur($i, "Nb entrées", $bo->nb_entrees);

      // Second tableau: présentation par titre et par semaine
      $tableauB->ajoutEntete(2, $bo->semaine, "Semaine " . $bo->
          semaine);
      $tableauB->ajoutValeur($bo->titre, $bo->semaine, $bo->
          nb_entrees);
  }

  // Affichage des tableaux
  echo $tableauA->tableauHTML() . "<br/>\n";
```

```
    echo $tableauB->tableauHTML() . "<br/>\n";
}
catch (Exception $exc) {
  // Une erreur est survenue
  echo "<b>Erreur rencontrée:</b> " . $exc->getMessage() . "\n";
}
?>
</body>
</html>
```



Film	Ville	Semaine	Nb entrées
Matrix	Paris	1	12000
Matrix	Paris	2	15000
Matrix	Paris	3	11000
Spiderman	Paris	1	8000
Spiderman	Paris	2	9000
Spiderman	Paris	3	9500

Box office	Semaine 1	Semaine 2	Semaine 3
Matrix	12000	15000	11000
Spiderman	8000	9000	9500

Figure 3.3 — Affichage des deux tableaux.

Bien entendu on utilise un objet de la classe `BDMySQL` pour se connecter, effectuer une requête et parcourir le résultat. Ce qui nous intéresse ici c'est la production des tableaux. Le premier, `tableauA`, est instancié comme suit :

```
$tableauA = new Tableau(2, array("border"=>2));
$tableauA->setAfficheEntete(1, FALSE);
```

On indique donc qu'il s'agit d'un tableau à deux dimensions, avec une bordure de 2 pixels. On peut noter la pratique consistant à passer un nombre variable de paramètres (ici des attributs HTML) sous la forme d'un tableau PHP. La seconde instruction supprime l'affichage des en-têtes de la dimension 1.

Ensuite, à chaque fois que la boucle sur le résultat de la requête renvoie un objet `bo`, on insère des valeurs avec la méthode `ajoutValeur()`. Rappelons que cette fonction définit la valeur de $M[c_1, c_2]$ où c_1 (respectivement c_2) est la clé désignant la ligne (respectivement la colonne) de la cellule.

```
$i++;
$tableauA->ajoutValeur($i, "Film", $bo->titre);
$tableauA->ajoutValeur($i, "Ville", $bo->ville);
$tableauA->ajoutValeur($i, "Semaine", $bo->semaine);
$tableauA->ajoutValeur($i, "Nb entrées", $bo->nb_entrees);
```

Ici la clé de la dimension 1 (les lignes) est basée sur un compteur incrémenté à chaque passage dans la boucle, et la clé de la dimension 2 (les colonnes) est un texte qui servira également d'en-tête (voir figure 3.3).

Pour le second tableau, `tableauB`, on applique les mêmes principes. L'instanciation est identique. On appelle deux méthodes qui fixent le libellé du coin supérieur gauche, et une couleur de fond pour les lignes impaires.

```
$tableauB->setCoinSuperieurGauche("Box office");
$tableauB->setCouleurImpaire("silver");
```

Puis, à chaque passage dans la boucle, on insère une valeur de la mesure `nbEntrées` indexée par le titre du film (dimension 1, les lignes) et par la semaine (dimension 2, les colonnes). De plus, au lieu de garder l'en-tête par défaut pour les colonnes (le numéro de la semaine), on le définit avec la méthode `ajoutEntete()` comme étant la concaténation de la chaîne `"Semaine "` et du numéro de semaine.

```
$tableauB->ajoutEntete(2, $bo->semaine, "Semaine " . $bo->semaine);
$tableauB->ajoutValeur($bo->titre ,  $bo->semaine, $bo->nbEntrees);
```

Il n'y a rien de plus à faire. L'appel de la méthode `tableauHTML()` renvoie une chaîne qui peut être placée dans un document HTML. Bien entendu on pourrait améliorer la présentation, par exemple en cadrant à droite les colonnes contenant des nombres. C'est possible – et facile- - en ajoutant des méthodes appropriées à la classe `Tableau`. Ce type d'extension est très utile à réaliser pour bien comprendre comment fonctionne une classe.

Cet exemple montre comment la programmation objet permet de s'affranchir de détails de bas niveau comme, ici, les balises HTML à utiliser en ouverture et en fermeture ou l'ordre de création des cellules. On se contente de déclarer le contenu du tableau et l'objet se charge de fournir une chaîne de caractères contenant sa description HTML. Cette chaîne peut alors être utilisée par l'application comme bon lui semble. On pourrait par exemple la placer dans une cellule d'un autre tableau pour obtenir très facilement des imbrications. Ce qui compte, pour bien utiliser la classe, c'est d'une part de comprendre la modélisation (et donc ce qu'est conceptuellement un objet tableau), et d'autre part de connaître les modes de contrôle et d'interaction avec l'objet.

3.2.3 Implantation

Il reste à regarder le code de la classe pour voir comment les différentes méthodes sont implantées. Rappelons que la consultation du code est *inutile* si on souhaite seulement *utiliser* la classe. D'ailleurs dans des langages compilés comme C++ et Java, le code n'est pas disponible ; seules les spécifications de l'interface sont fournies aux utilisateurs.

Le code de la classe `tableau` est bien entendu disponible sur le site de ce livre. Nous allons présenter les parties les plus importantes, en les commentant à chaque fois. Pour commencer, on trouve les propriétés, toutes privées.

```
class Tableau
{
  // ——— Partie privée : les constantes et les variables
  private $nb_dimensions;
  // Tableau des valeurs à afficher
  private $tableau_valeurs;
  // Tableaux des en-têtes
  private $entetes, $options_lig, $options_col;
  // Options de présentation pour la table. A compléter.
  private $options_tables, $couleur_paire, $couleur_impaire,
    $csg, $affiche_entete, $repetition_ligne=array(),
    $option_dim=array();
  // Constante pour remplir les cellules vides
  const VAL_DEFAUT=" ";
```

On trouve la dimension du tableau, le tableau des valeurs ($M[c_1][c_2]$ dans la modélisation) et le tableau des en-têtes ($e[d, c]$ dans la modélisation). Les autres attributs sont tous destinés à la présentation HTML. Une nouveauté syntaxique, non rencontrée jusqu'à présent, est la définition d'une constante locale à la classe, qui peut être référencée avec la syntaxe `self::VAL_DEFAUT` ou `Tableau::VAL_DEFAUT`.

Le constructeur, donné ci-dessous, effectue essentiellement des initialisations. Il manque de nombreux tests pour améliorer la robustesse de la classe. Je vous invite à y réfléchir et ajouter les contrôles et levées d'exceptions nécessaires (ne faudrait-il pas par exemple s'inquiéter des valeurs possibles de la dimension ?).

```
  function __construct ($nb_dimensions=2, $tab_attrs=array())
  {
    // Initialisation des variables privées
    $this->tableau_valeurs = array();
    $this->options_tables=$this->couleur_paire=$this->
        couleur_impaire="";

    // Initialisation de la dimension. Quelques tests s'imposent
    // ...
    $this->nb_dimensions=$nb_dimensions;

    // Initialisation des tableaux d'en-têtes pour chaque
        dimension
    for ($dim=1; $dim <= $this->nb_dimensions; $dim++) {
```

```
        $this->entetes[$dim] = array();
            $this->affiche_entete[$dim] = TRUE;
    }
    // Attributs de la balise <table>
    $this->ajoutAttributsTable($tab_attrs);
}
```

Les méthodes commençant `set` ou `get`, traditionnelles en programmation objet, ne servent à rien d'autre le plus souvent qu'à accéder, en écriture ou en lecture, aux propriétés de la classe (on parle parfois d'« accesseurs »). En voici un exemple avec la méthode `setCouleurImpaire()` qui affecte la couleur de fond des lignes impaires.

```
public function setCouleurImpaire($couleur)  {
  $this->couleurImpaire = $couleur;
}
```

Bien entendu, il suffirait de rendre publique la propriété `couleur_impaire` pour éviter d'écrire une méthode spéciale. Les scripts pourraient alors directement la modifier. Cela rendrait malheureusement définitivement impossible toute évolution ultérieure pour contrôler la valeur affectée à `couleur_impaire`. Plus généralement, rendre publique une propriété empêche toute modification ultérieure apportée à l'organisation interne d'une classe.

> **REMARQUE –** PHP 5 fournit des méthodes dites « magiques » pour éviter la programmation systématique des accesseurs. La méthode `__get(`*nom*`)` est appelée chaque fois que l'on utilise la syntaxe `$o->nom` pour *lire* une propriété qui n'existe pas explicitement dans la classe ; `__set(`*nom*`, `*valeur*`)` est appelée quand on utilise la même syntaxe pour faire une affectation. Enfin, `__call(`*nom*`, `*params*`)` intercepte tous les appels à une méthode qui n'existe pas. Des exemples de ces méthodes sont donnés page 267.

La méthode `ajoutValeur()` insère une nouvelle valeur dans une cellule dont les coordonnées sont données par les deux premiers paramètres. Voici son code. Notez qu'on en profite pour affecter une valeur par défaut (la valeur de la clé elle-même) à l'en-tête de la ligne et de la colonne correspondante. Ici encore quelques contrôles (par exemple sur les paramètres en entrée) seraient les bienvenus.

```
public function ajoutValeur($cle_ligne, $cle_colonne, $valeur)
{
  // Maintenance des en-têtes
  if (!array_key_exists($cle_ligne, $this->entetes[1]))
    $this->entetes[1][$cle_ligne] = $cle_ligne;
  if (!array_key_exists($cle_colonne, $this->entetes[2]))
    $this->entetes[2][$cle_colonne] = $cle_colonne;

  // Stockage de la valeur
  $this->tableau_valeurs[$cle_ligne][$cle_colonne] = $valeur;
}
```

Le code donné ci-dessus fonctionne pour les tableaux à deux dimensions. Pour les tableaux de dimension quelconque, l'implantation est un peu plus compliquée, mais figure dans le code fourni sur le site.

Troisième méthode importante, `ajoutEntete()` se contente d'affecter un texte à l'en-tête d'une ligne ou d'une colonne (selon la dimension passée en paramètre) pour une valeur de clé donnée. Comme on l'a vu ci-dessus, par défaut cet en-tête sera la clé elle-même, ce qui peut convenir dans beaucoup de cas.

```
public function ajoutEntete($dimension, $cle, $texte)
{
  // Stockage de la chaîne servant d'en-tête
  $this->entetes[$dimension][$cle] = $texte;
}
```

Il reste finalement (en ignorant d'autres méthodes annexes que je vous laisse consulter directement dans le code) la méthode produisant le tableau HTML. Partant de toutes les mesures reçues au fur et à mesure et stockées dans les propriétés d'un objet, cette méthode construit une chaîne de caractères contenant les balises HTML appropriées. Le code ci-dessous est une version légèrement simplifiée de la méthode complète.

```
 function tableauHTML()
{
  $chaine = $ligne = "";

  // Affiche-t'on le coin supérieur gauche?
  if ($this->affiche_entete[1]) $ligne = "<th>$this->csg</th>";

  if (!empty($this->legende)) {
      $nb_cols = count($this->entetes[2]);
      $chaine = "<tr class='header'>\n<th colspan=$nb_cols>
          $this->legende"
        . "</th>\n</tr>\n";
  }

  // Création des ent-êtes de colonnes (dimension 2)
  if ($this->affiche_entete[2]) {
      foreach ($this->entetes[2] as $cle => $texte)
        $ligne .= "<th>$texte</th>\n";

      // Ligne des en-têtes.
      $chaine = "<tr class='header'>$ligne</tr>\n";
  }

  $i=0;
  // Boucles imbriquées sur les deux tableaux de clés
  foreach ($this->entetes[1] as $cle_lig => $enteteLig) //
      Lignes
  {
    if ($this->affiche_entete[1])
        $ligne = "<th>$enteteLig</th>\n";
      else
        $ligne = "";

      $i++;
```

```
        foreach ($this->entetes[2] as $cle_col => $enteteCol) //
            Colonnes
          {
            // On prend la valeur si elle existe, sinon le défaut
            if (isSet($this->tableau_valeurs[$cle_lig][$cle_col])
                )
              $valeur = $this->tableau_valeurs[$cle_lig][$cle_col];
            else
              $valeur = self::VAL_DEFAUT;

            // On place la valeur dans une cellule
            $ligne .= "<td>$valeur</td>\n";
          }
        // Eventuellement on tient compte de la couleur
        if ($i % 2 == 0) {
            $options_lig = " class='even'";
            if (!empty($this->couleur_paire))
              $options_lig .= " bgcolor='$this->couleur_paire' ";
        }
        else if ($i % 2 == 1) {
            $options_lig = " class='odd'";
            if (!empty($this->couleur_impaire))
              $options_lig = " bgcolor='$this->couleur_impaire' ";
        }
        else $options_lig = "";

        // Doit-on appliquer une option?
        if (isSet($this->options[1][$cle_lig]))
          foreach ($this->options[1][$cle_lig] as $option =>
              $valeur)
            $options_lig .= " $option='$valeur' ";
        $ligne = "<tr$options_lig>\n$ligne\n</tr>\n";

        // Prise en compte de la demande de répétition d'une
        // ligne
        if (isSet($this->repetition_ligne[1][$cle_lig])) {
            $rligne = "";
            for ($i=0; $i < $this->repetition_ligne[1][$cle_lig];
                 $i++)
              $rligne .= $ligne;
            $ligne = $rligne;
        }
        // On ajoute la ligne à la chaîne
        $chaine .= $ligne;
      }
    // Placement dans la balise TABLE, et retour
    return  "<table $this->options_tables>\n$chaine</table>\n";
}
```

Le tableau associatif `entetes` contient toutes les clés permettant d'accéder aux cellules stockées dans le tableau `tableau_valeurs`, ainsi que les libellés associés à ces clés et utilisés pour les en-têtes. Il suffit donc d'une boucle sur chaque dimension pour récupérer les coordonnées d'accès à la cellule, que l'on peut alors insérer dans des balises HTML. Pour le tableau B par exemple, le contenu du tableau `entetes`, tel qu'on peut le récupérer avec la fonction PHP `print_r()` qui affiche tous les éléments d'un tableau, est le suivant :

- dimension 1 :
  `Array ( [Matrix] => Matrix [Spiderman] => Spiderman )`
- dimension 2 :
  `Array ( [1] => Semaine 1 [2] => Semaine 2 [3] => Semaine 3 )`

En parcourant les *clés* à l'aide des boucles imbriquées de la méthode `tableauHTML()`, on obtient les paires `(Matrix, 1)`, `(Matrix, 2)`, `(Matrix, 3)`, puis `(Spiderman, 1)`, `(Spiderman, 2)`, `(Spiderman, 3)`. Chaque paire `(clé1, clé2)` définit une entrée `tableauValeurs[cle1][cle2]`.

3.3 LA CLASSE FORMULAIRE

Voici un deuxième exemple de classe « utilitaire » visant à produire du code HTML complexe, en l'occurrence une classe `Formulaire` pour générer des formulaires HTML. Outre la création des champs de saisie, cette classe permet de soigner la présentation des formulaires en alignant les champs et les textes explicatifs à l'aide de tableaux HTML[3]. Comme précédemment, nous cherchons à obtenir des fonctionnalités puissantes par l'intermédiaire d'une interface la plus simple possible, en cachant donc au maximum la complexité du code.

3.3.1 Conception

Comme pour la classe `Tableau`, il faut fixer précisément le type de service qui sera fourni par la classe en cherchant un bon compromis entre les fonctionnalités, et la complexité de leur utilisation.

Le premier rôle de la classe est de permettre la création de champs de saisie, conformes à la spécification HTML, avec toutes les options possibles : taille affichée, taille maximale, valeur par défaut et même contrôles Javascript. Un objet de la classe devra donc servir « d'usine » à fabriquer ces champs en utilisant des méthodes dédiées auxquelles on passe les paramètres appropriés. De plus chaque champ doit pouvoir être accompagné d'un libellé indiquant sa destination. On pourra par exemple disposer d'une méthode de création d'un champ de saisie de texte :

champTexte (*libellé*, *nomChamp*, *valeurDéfaut*, *tailleAffichée*, *tailleMax*)

3. Il est également possible d'obtenir cet alignement avec des feuilles de style CSS.

Par ailleurs la classe `Formulaire` doit également fournir des fonctionnalités de placement des champs et des libellés les uns par rapport aux autres, ce qui peut se gérer à l'aide de tableaux HTML.

La figure 3.4 montre les possibilités attendues, avec des traits en pointillés qui indiquent le tableau HTML permettant d'obtenir un alignement régulier des différents composants du formulaire. La première partie présente les champs dans un tableau à deux colonnes, la première correspondant aux libellés, et la seconde aux champs quel que soit leur type : texte, mot de passe, liste déroulante, etc. Dans le cas où le champ consiste en un ensemble de choix matérialisés par des boutons (le champ 4 dans la figure), on souhaite créer une table imbriquée associant à chaque bouton un sous-libellé, sur deux lignes. Ce premier type de présentation sera désigné par le terme *Mode Table, orientation verticale*.

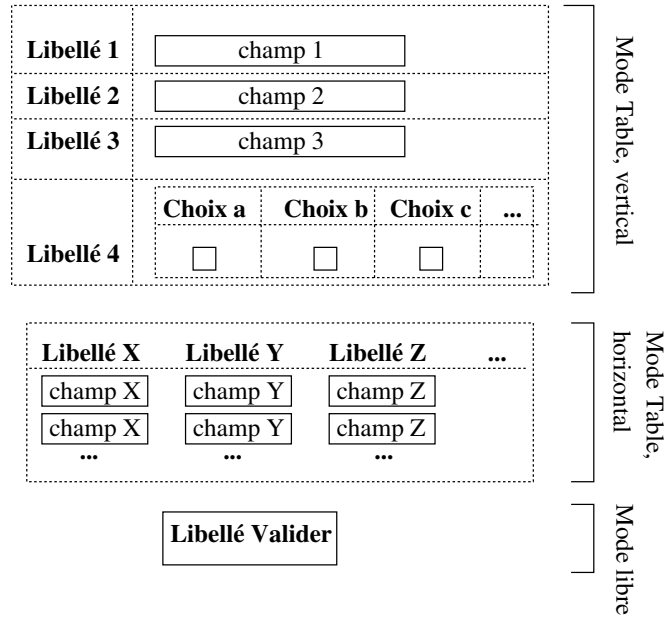


Figure 3.4 — Conception de la classe `Formulaire`

La deuxième partie du formulaire de la figure 3.4 organise la présentation en autant de colonnes qu'il y a de champs, ces colonnes étant préfixées par le libellé du champ. Ce mode de présentation, désigné par le terme *Mode Table, orientation horizontale*, permet d'affecter plusieurs zones de saisie pour un même champ, une par ligne.

Enfin, le formulaire doit permettre une présentation libre – c'est-à-dire sans alignement à l'aide de tableau – comme le bouton de validation à la fin du formulaire.

La classe doit proposer un ensemble de méthodes pour créer tous les types de champs possibles dans un formulaire HTML, et disposer chaque champ en fonction du mode de présentation qui a été choisi. Cela suppose que l'objet chargé de produire

le formulaire connaisse, lors de la création du champ, le mode de présentation courant.

En résumé il s'agit d'implanter une fois pour toutes, sous forme de classe orientée-objet, les tâches courantes de production et de mise en forme de formulaire que l'on trouve dans toutes les applications web en général, et tout particulièrement dans les applications s'appuyant sur une base de données.

3.3.2 Utilisation

Commençons par présenter l'utilisation de la classe avant d'étudier ses mécanismes internes. La liste des méthodes publiques est donnée dans la table 3.8. Elles appartiennent à deux catégories :

Tableau 3.8 — Les méthodes publiques de la classe `Formulaire`

Méthode	Description
champTexte(*libellé, nom, val, long, longMax*)	Champ de saisie de texte.
champMotDePasse(*libellé, nom, val, long, longMax*)	Champ de saisie d'un mot de passe.
champRadio(*libellé, nom, val, liste*)	Boutons `radio`
champListe(*libellé, nom, val, taille, liste*)	Boutons `select`
champFenetre(*libellé, nom, val, ligs, cols*)	Boutons `textarea`
champCache(*nom, val*)	Champ caché.
champFichier(*libellé, nom, taille*)	Champ fichier (*upload*).
champValider(*libellé, nom*)	Bouton `submit`
debutTable(*orientation, attributs, nbLignes*)	Entrée en mode table.
ajoutTexte(*texte*)	Ajout d'un texte libre.
finTable()	Sortie du mode table.
getChamp(*idChamp*)	Récupération d'un champ du formulaire.
formulaireHTML()	Retourne la chaîne de caractères contenant le formulaire HTML.

- *Production d'un champ de formulaire.*
  À chaque type de champ correspond une méthode qui ne prend en argument que les paramètres strictement nécessaires au type de champ souhaité. Par exemple la méthode `champTexte()` utilise un libellé, le nom du champ, sa valeur par défaut, sa taille d'affichage et la taille maximale (ce dernier paramètre étant optionnel). Parmi les autres méthodes, on trouve `champRadio()`, `champListe()`, `champFenetre()`, etc.
  Chaque méthode renvoie l'identifiant du champ créé. Cet identifiant permet d'accéder au champ, soit pour le récupérer et le traiter isolément (méthode `getChamp()`), soit pour lui associer des contrôles Javascript ou autres.
- *Passage d'un mode de présentation à un autre.*
  Ces méthodes permettent d'indiquer que l'on entre ou sort d'un mode Table, en horizontal ou en vertical.

Voici un premier exemple illustrant la simplicité de création d'un formulaire. Il s'agit d'une démonstration des possibilités de la classe, sans déclenchement d'aucune action quand le formulaire est soumis. Nous verrons dans le chapitre 5 comment utiliser cette classe en association avec la base de données pour créer très rapidement des interfaces de saisie et de mise à jour.

Exemple 3.9 *exemples/ApplClasseFormulaire.php* : *Exemple démontrant les possibilités de la classe* `Formulaire`.

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Création d'un formulaire</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>

<?php
require_once ("Formulaire.php");

// Instanciation du formulaire
$form = new Formulaire ("post", "ApplClasseFormulaire.php");

// Un champ caché
$form->champCache ("mode", "Démonstration");

// Tableau en mode vertical, avec quelques champs
$form->debutTable(Formulaire::VERTICAL);
$form->champTexte ("Nom", "nom", "Entrez votre nom", 40);
$form->champTexte ("Prénom", "prenom", "Entrez votre prénom", 40);

// Un champ radio, avec la liste des choix dans un tableau PHP
$form->champRadio ("Sexe", "sexe", "M", array("M" => "Masculin",
                                              "F"=>"Féminin"));

// Un champ select, avec la liste des choix dans un tableau PHP
$form->champListe ("Justificatif", "nation", "cni", 1,
                   array( "cni"=>"Carte d'identité",
                          "pass"=>"Passeport",
                          "pc"=>"Permis de conduire"));
// Un champ textarea
$form->champFenetre ("Bref CV", "cv", "Votre CV en quelques
    lignes", 4, 50);
// Un champ fichier
$form->champFichier ("Votre photo", "photo", 30);

// Fin du mode vertical
$form->finTable();
```

```
$form->ajoutTexte ("<b>Vos enfants</b>");

// Tableau en mode horizontal, avec 5 lignes
$form->debutTable (Formulaire::HORIZONTAL, array(), 5);
$form->champTexte ("Prénom", "prenom[]", "", 20, 30);
$form->champTexte ("Nom", "nom[]", "", 20, 30);
$form->champTexte ("Né en", "annee_naissance[]", "", 4);
$form->finTable ();

// Bouton de validation, avec placement libre
$form->champValider ("Valider la saisie", "valider");

// Affichage du formulaire
echo $form->formulaireHTML();

?>
```

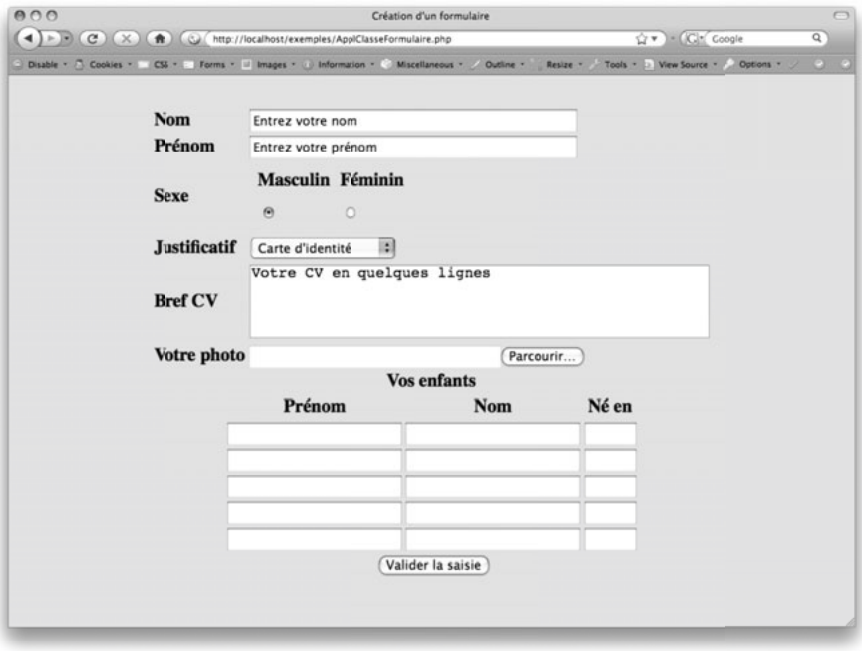


Figure 3.5 — Affichage du formulaire de démonstration.

L'affichage du formulaire est donné dans la figure 3.5. Quelques lignes de spécification, accompagnées du nombre strictement minimal de paramètres, suffisent pour créer ce formulaire, sans qu'il soit nécessaire d'avoir à produire explicitement la moindre balise HTML. Cet avantage est d'autant plus appréciable que le résultat comprend une imbrication assez complexe de balises de formulaires, de tableaux, et de données provenant du script PHP, qui seraient très fastidieuses à intégrer si l'on ne disposait pas de ce type d'outil automatisé.

3.3.3 Implantation

L'implantation de la classe nécessite des structures internes un peu plus sophistiquées que celles vues jusqu'à présent. Il s'agit en effet de décrire le contenu d'un formulaire, sous une forme offrant le plus de souplesse possible. On doit être capable par exemple de récupérer individuellement la description HTML de l'un des champs, ou de désigner un champ auquel on souhaite associer un contrôle Javascript. Enfin les propriétés doivent contenir toutes les informations nécessaires pour produire la chaîne HTML du formulaire.

On va représenter ce contenu sous la forme d'une liste de *composants*, à choisir parmi

- un champ de saisie, accompagné de son libellé ;
- un texte libre à insérer dans le formulaire ;
- l'indication d'un début de tableau, accompagné des caractéristiques du tableau ;
- l'indication d'une fin de tableau.

La figure 3.6 montre l'organisation globale de la classe, en distinguant la partie publique (en haut) proposant une interface à l'utilisateur, et une partie privée (en bas) constituée de méthodes internes et de propriétés (essentiellement, ici, les composants). Le principe général est que toutes les méthodes insèrent de nouveaux composants, sauf *formulaireHTML()* qui va consulter les composants existants pour produire le formulaire.

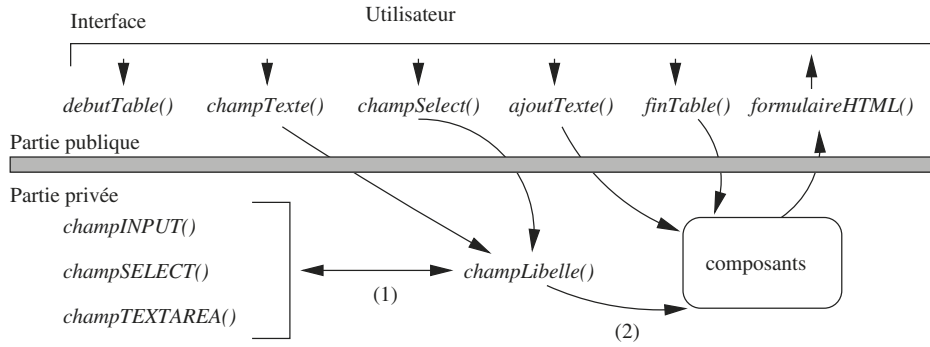


Figure 3.6 — Organisation de la classe `Formulaire`

> **REMARQUE –** On pourrait (devrait ...) créer une classe `FormComposant` pour représenter et manipuler ces composants. En programmation objet, tout concept doit donner lieu à la création d'une classe, avec des avantages à moyen et long terme en matière d'évolutivité. L'inconvénient est de rendre la conception et l'organisation des classes plus ardu à maîtriser. C'est la raison pour laquelle nous n'allons pas plus loin, au moins dans ce chapitre.

L'insertion d'un nouveau composant se fait directement pour le début ou la fin d'une table, et pour l'ajout d'un texte. Pour l'ajout d'un champ accompagné de son

libellé, on a recours à un ensemble de méthodes privées un peu plus important. Tout d'abord, il existe une méthode dédiée à chaque type de champ (`input`, `select`, etc.) qui se charge de construire la balise HTML correctement. Ensuite toute demande de création d'un champ passe par la méthode *champLibelle()* qui choisit d'abord (flèche 1), en fonction de la demande, la méthode de création de champ spécialisée, puis crée le composant avec le champ et le libellé (flèche 2).

Voyons maintenant dans le détail le code de la classe, en commençant par les propriétés.

```
class Formulaire
{
 // ----   Partie privée : les propriétés et les constantes
 const VERTICAL = 1;
 const HORIZONTAL = 2;

 // Propriétés de la balise <form>
 private $methode, $action, $nom, $transfertFichier=FALSE;

 // Propriétés de présentation
 private  $orientation="", $centre=TRUE, $classeCSS, $tableau ;

 // Propriétés stockant les composants du formulaire
 private $composants=array(), $nbComposants=0;
```

On trouve donc :

- les paramètres à placer dans la balise ouvrante <**form**>, avec `methode` qui peut valoir `get` ou `post`, `action`, le nom du script à déclencher sur validation du formulaire, `transfertFichier` qui indique si le formulaire permet ou non de transférer des fichiers, et le nom du formulaire qui peut être utilisé pour les contrôles Javascript ;
- les propriétés déterminant la présentation du formulaire, avec `orientation`, qui peut être soit `VERTICAL`, soit `HORIZONTAL`, deux constantes locales à la classe, soit la chaîne vide qui indique qu'on n'est pas en mode table. La variable booléenne `centre` indique si le formulaire doit être centré dans la page HTML. Enfin une variable `tableau`, correspondant à un objet de la classe `Tableau` qui va nous aider à mettre en forme les champs ;
- la représentation des composants : un simple tableau, et le nombre de composants créés à un moment donné.

Bien entendu, comme toutes les classes objet, celle-ci ne demande qu'à être complétée. Des suggestions en ce sens sont proposées dans le polycopié d'exercices.

Constructeur

Le constructeur de la classe `Formulaire` se contente d'initialiser les attributs, notamment ceux qui seront par la suite placés dans la balise ouvrante <**form**>. Deux d'entre eux sont obligatoires : la méthode employée (qui est en général `post`)

et le nom du script associé au formulaire. Les paramètres optionnels indiquent si le formulaire doit être centré, la classe CSS définissant la présentation (cette classe n'est pas utilisée dans la version présentée ici), et le nom du formulaire.

```
function Formulaire ($methode, $action, $centre=true,
                     $classe="Form", $nom="Form")
{
  // Initialisation des propriétés de l'objet avec les
     paramètres
  $this->methode = $methode;
  $this->action = $action;
  $this->classeCSS = $classe;
  $this->nom = $nom;
  $this->centre = $centre;
}
```

Quelques contrôles seraient les bienvenus (sur la méthode par exemple, qui ne peut prendre que deux valeurs). Comme d'habitude nous les omettons pour ne pas surcharger le code.

Méthodes privées

La classe comprend ensuite un ensemble de méthodes privées pour produire les champs d'un formulaire HTML. Toutes ces méthodes renvoient une chaîne de caractères contenant la balise complète, prête à insérer dans un document HTML. La méthode `champINPUT()`, ci-dessous, produit par exemple un champ `input` du type demandé, avec son nom, sa valeur, le nombre de caractères du champ de saisie, et le nombre maximal de caractères saisissables par l'utilisateur[4].

```
// Méthode pour créer un champ input général
private function champINPUT ($type, $nom, $val, $taille,
    $tailleMax)
{
  // Attention aux problèmes d'affichage
  $val = htmlSpecialChars($val);

  // Création et renvoi de la chaîne de caractères
  return "<input type='$type' name=\"$nom\" "
    ."value=\"$val\" size='$taille' maxlength='$tailleMax'/>\n";
}
```

Quand on manipulate des chaînes en y incluant des variables, attention à bien imaginer ce qui peut se passer si les variables contiennent des caractères gênants comme « ' ». Pour l'attribut `value` par exemple, on a appliqué au préalable la fonction `htmlSpecialChars()`.

Les paramètres passés aux méthodes créant des champs peuvent varier en fonction du type de champ produit. Par exemple les méthodes produisant des listes de choix

4. On peut faire défiler un texte dans un champ de saisie. Le nombre de caractères saisissables n'est donc pas limité par la taille d'affichage du champ.

prennent en argument un tableau associatif – comme `$liste` dans la méthode ci-dessous – dont la clé est la valeur de chaque choix, et l'élément le libellé associé.

```
// Champ pour sélectionner dans une liste
private function  champSELECT ($nom, $liste, $defaut, $taille
    =1)
{
  $s = "<select name=\"$nom\" size='$taille'>\n";
  while (list ($val, $libelle) = each ($liste)) {
    // Attention aux problèmes d'affichage
        $val = htmlSpecialChars($val);
        $defaut = htmlSpecialChars($defaut);

    if ($val != $defaut)
     $s .=  "<option value=\"$val\">$libelle</option>\n";
    else
     $s .= "<option value=\"$val\" selected='1'>$libelle</
         option>\n";
    }
  return $s . "</select>\n";
}
```

Dans la méthode ci-dessus, on crée un champ `select` constitué d'une liste d'options. Une autre méthode `champBUTTONS`, que nous vous laissons consulter dans le fichier *Formulaire.php*, dispose tous les choix sur deux lignes, l'une avec les libellés, l'autre avec les boutons correspondants. Elle est utilisée pour les listes de boutons `radio` ou `checkbox`.

Une méthode plus générale permet de produire la chaîne contenant un champ de formulaire, quel que soit son type. Comme les paramètres peuvent varier selon ce type, on utilise à cette occasion une astuce de PHP pour passer un nombre variable de paramètres. La variable `$params` ci-dessous est un tableau associatif dont la clé est le nom du paramètre, et l'élément la valeur de ce paramètre. Dans le cas d'un champ `textarea` par exemple, `$params` doit être un tableau à deux éléments, l'un indexé par `ROWS` et l'autre par `COLS`.

```
// Champ de formulaire
 private function champForm ($type, $nom, $val, $params, $liste=
     array())
 {
   switch ($type)
   {
     case "text": case "password": case "submit": case "reset":
     case "file": case "hidden":
       // Extraction des paramètres de la liste
       if (isSet($params['SIZE']))
       $taille = $params["SIZE"];
       else  $taille = 0;
       if (isSet($params['MAXLENGTH']) and $params['MAXLENGTH'
           ]!=0)
       $tailleMax = $params['MAXLENGTH'];
       else $tailleMax = $taille;
```

```
            // Appel de la méthode champInput
            $champ = $this->champInput ($type, $nom, $val, $taille,
                $tailleMax);
            // Si c'est un transfert de fichier: s'en souvenir
            if ($type == "file") $this->transfertFichier=TRUE;
            break;

        case "textarea":
            $lig = $params["ROWS"]; $col = $params["COLS"];
            // Appel de la méthode champTextarea de l'objet courant
            $champ = $this->champTextarea ($nom, $val, $lig, $col);
            break;

        case "select":
            $taille = $params["SIZE"];
            // Appel de la méthode champSelect de l'objet courant
            $champ = $this->champSelect ($nom, $liste, $val, $taille)
                ;
            break;

        case "checkbox":
            $champ = $this->champButtons ($type, $nom, $liste, $val,
                $params);
            break;

        case "radio":
            // Appel de la méthode champButtons de l'objet courant
            $champ = $this->champButtons ($type, $nom, $liste, $val,
                array());
            break;

        default: echo "<b>ERREUR: $type est un type inconnu</b>\n";
            break;
    }
    return $champ;
}
```

Quand un bouton `file` est créé, on positionne la propriété `$this->transfertFichier` à `true` pour être sûr de bien produire la balise <**form**> avec les bons attributs. En fait, avec cette technique, on est assuré que le formulaire sera *toujours* correct, sans avoir à s'appuyer sur le soin apporté au développement par l'utilisateur de la classe.

La méthode `champForm` permet d'appeler la bonne méthode de création de champ en fonction du type souhaité. La structure de test `switch` utilisée ci-dessus (voir chapitre 11) est bien adaptée au déclenchement d'une action parmi une liste prédéterminée en fonction d'un paramètre, ici le type du champ. L'ensemble des types de champ `text`, `password`, `submit`, `reset`, `hidden` et `file` correspond par exemple à la méthode `champInput`.

Création des composants

Finalement nous disposons de tous les éléments pour commencer à construire les composants d'un formulaire. Chacune des méthodes qui suit construit un composant et le stocke dans la propriété `composants` de l'objet. Voici le cas le plus simple, pour commencer : l'ajout d'un composant de texte dans le formulaire.

```
// Ajout d'un texte quelconque
public function ajoutTexte ($texte)
{
  // On ajoute un élément dans le tableau $composants
  $this->composants[$this->nbComposants] = array("type"=>"TEXTE",
                                                 "texte" => $texte);
  // Renvoi de l'identifiant de la ligne, et incrémentation
  return $this->nbComposants++;
}
```

Un composant est représenté par un tableau associatif PHP comprenant *toujours* un élément `type`, et d'autres éléments qui dépendent du type de composant. Pour un texte, on a simplement un élément `texte` mais nous verrons que pour un champ la description est un peu plus riche.

Le composant est stocké dans le tableau `composants`, propriété de l'objet, et indicé par un numéro qui tient lieu d'identifiant pour le composant. Cet identifiant est renvoyé de manière à ce que l'utilisateur garde un moyen de référencer le composant pour, par exemple, le récupérer ou le modifier par l'intermédiaire d'autres méthodes.

La seconde méthode (privée celle-là) construisant des composants est `champLibelle()`.

```
  // Création d'un champ avec son libellé
  private function champLibelle ($libelle, $nom, $val, $type,
                          $params=array(), $liste=array())
  {
   // Création de la balise HTML
   $champHTML = $this->champForm ($type, $nom, $val, $params,
        $liste);

   // On met le libellé en gras
   $libelle = "<b>$libelle</b>";

   // Stockage du libellé et de la balise dans le contenu
   $this->composants[$this->nbComposants] = array("type" => "CHAMP",
                                                  "libelle" => $libelle,
                                                  "champ" => $champHTML);

   // Renvoi de l'identifiant de la ligne, et incrémentation
   return $this->nbComposants++;
  }
```

La méthode `champLibelle()` construit tout d'abord, par appel à la méthode `champForm()`, une chaîne contenant le champ du formulaire. On dispose alors des variables `$champHTML` et `$pLibelle` que l'on place simplement dans le tableau représentant le composant, en indiquant que le type de ce dernier est `CHAMP`.

Le troisième type de composant à créer indique le début d'un tableau permettant d'afficher les champs de manière ordonnée.

```
// Début d'une table, mode horizontal ou vertical
public function debutTable ($orientation=Formulaire::VERTICAL,
                $attributs=array(), $nbLignes=1)
{
  // On instancie un objet pour créer ce tableau HTML
  $tableau = new Tableau (2, $attributs);

  // Jamais d'affichage de l'en-tête des lignes
  $tableau->setAfficheEntete (1, FALSE);

  // Action selon l'orientation du tableau
  if ($orientation == Formulaire::HORIZONTAL)
     $tableau->setRepetitionLigne (1, "ligne", $nbLignes);
  else // Pas d'affichage non plus de l'en-tête des colonnes
     $tableau->setAfficheEntete (2, FALSE);

  // On crée un composant dans lequel on place le tableau
  $this->composants[$this->nbComposants] =
                        array("type"=>"DEBUTTABLE",
                              "orientation"=> $orientation,
                              "tableau"=> $tableau);

  // Renvoi de l'identifiant de la ligne et incrémentation
  return $this->nbComposants++;
}
```

La présentation basée sur un tableau est, bien entendu, déléguée à un objet de la classe `Tableau` (voir section précédente) spécialisé dans ce type de tâche. On instancie donc un objet de cette classe quand on sait qu'il faudra produire un tableau, et on le configure selon les besoins de mise en forme du formulaire (revoir si nécessaire la figure 3.4, page 153, pour la conception de la classe et les règles de présentation). Ici :

1. quelle que soit l'orientation, horizontale ou verticale, on n'utilise jamais d'en-tête pour les lignes ;
2. en affichage horizontal, on répète *n* fois la ligne contenant les différents champs en appelant la méthode *repetitionLigne()* de la classe `Tableau` (non présentée précédemment, mais consultable dans le code) ;
3. en affichage vertical, on n'affiche pas non plus d'en-tête pour les colonnes.

Une fois configuré, l'objet tableau est inséré, avec l'orientation choisie, dans le composant de type `DEBUTTABLE`. L'objet sera utilisé dès que l'on demandera la production du formulaire avec la méthode *formulaireHTML()*.

Enfin, voici la dernière méthode créant un composant marquant la fin d'une présentation basée sur un tableau HTML.

```
public function finTable ()
{
  // Insertion d'une ligne marquant la fin de la table
  $this->composants[$this->nbComposants++] = array("type"=>"
      FINTABLE");
}
```

Méthodes publiques de création de champs

Nous en arrivons à la partie publique de la classe correspondant à la création de champs. À titre d'exemple, voici trois de ces méthodes.

```
public function champTexte($libelle, $nom, $val, $taille,
    $tailleMax=0)
{
  return $this->champLibelle($libelle, $nom, $val,
                             "text", array("SIZE"=>$taille,
                                           "MAXLENGTH"=>$tailleMax));
}

public function champRadio($libelle, $nom, $val, $liste)
{
  return $this->champLibelle($libelle, $nom, $val,
                              "radio", array (), $liste);
}

public function champFenetre($libelle, $nom, $val, $lig, $col)
{
  return $this->champLibelle($libelle, $nom, $val, "textarea",
                             array("ROWS"=>$lig,"COLS"=>$col));
}
```

Toutes font appel à la même méthode privée `champLibelle()`, et renvoient l'identifiant de champ transmis en retour par cette dernière. La méthode `champLibelle()` est plus générale mais plus difficile d'utilisation. Le rôle des méthodes publiques est véritablement de servir *d'interface* aux méthodes privées, ce qui signifie d'une part restreindre le nombre et la complexité des paramètres, et d'autre part contrôler la validité des valeurs de ces paramètres avant de les transmettre aux méthodes privées. On pourrait contrôler par exemple que le nombre de colonnes d'un champ `textarea` est un entier positif.

Notez, dans l'appel à `champLibelle()`, le passage du cinquième paramètre sous forme d'un tableau associatif indiquant un des attributs de la balise HTML correspondante. Par exemple `champTexte()`, qui correspond à une balise <**input**>, indique la taille d'affichage et la taille maximale de saisie en passant comme paramètre `array("SIZE"=>$pTaille, "MAXLENGTH"=>$pTailleMax)`. Cette technique de

passage de paramètres est un peu délicate à utiliser car il est facile d'y introduire des erreurs. En s'en servant uniquement dans le cadre d'une méthode privée, on limite les inconvénients.

Production du formulaire

Finalement, il reste à produire le formulaire avec la méthode *formulaireHTML()*. Quand cette méthode est appelée, toute la description du contenu du formulaire est disponible dans le tableau `composants`. Il s'agit donc essentiellement de parcourir ces composants et de créer la présentation appropriée. On concatène alors successivement la balise d'ouverture, en faisant attention à utiliser l'attribut `enctype` si un champ de type `file` a été introduit dans le formulaire, puis la chaîne de caractères dans laquelle on a placé tous les composants, enfin la balise fermante.

```
public function formulaireHTML ()
{
  // On place un attribut enctype si on transfère un fichier
  if ($this->transfertFichier)
   $encType = "enctype='multipart/form-data'";
  else
   $encType="";

  $formulaire = "";
  // Maintenant, on parcourt les composants et on crée le HTML
  foreach ($this->composants as $idComposant => $description)
    {
      // Agissons selon le type de la ligne
      switch ($description["type"])
        {
        case "CHAMP":
          // C'est un champ de formulaire
          $libelle = $description['libelle'];
          $champ = $description['champ'];
          if ($this->orientation == Formulaire::VERTICAL)
            {
              $this->tableau->ajoutValeur($idComposant,
                  "libelle", $libelle);
              $this->tableau->ajoutValeur($idComposant, "champ",
                 $champ);
            }
          else if ($this->orientation == Formulaire::HORIZONTAL)
            {
              $this->tableau->ajoutEntete(2, $idComposant,
                  $libelle);
              $this->tableau->ajoutValeur("ligne", $idComposant,
                   $champ);
            }
          else
            $formulaire .= $libelle . $champ;
          break;
```

```
            case "TEXTE":
              // C'est un texte simple à insérer
              $formulaire .= $description['texte'];
              break;

            case "DEBUTTABLE":
              // C'est le début d'un tableau HTML
              $this->orientation = $description['orientation'];
              $this->tableau = $description['tableau'];
              break;

            case "FINTABLE":
              // C'est la fin d'un tableau HTML
              $formulaire .= $this->tableau->tableauHTML();
              $this->orientation="";
              break;

            default: // Ne devrait jamais arriver...
              echo "<p>ERREUR CLASSE FORMULAIRE!!</p>";
            }
        }

        // Encadrement du formulaire par les balises
        $formulaire = "\n<form  method='$this->methode' " . $encType
          . "action='$this->action' name='$this->nom'>"
          . $formulaire . "</form>";

        // Il faut éventuellement le centrer
        if ($this->centre) $formulaire = "<center>$formulaire
                                          </center>\n";;

        // On retourne la chaîne de caractères contenant le
        // formulaire
        return $formulaire;
    }
```

Essentiellement le code consiste à parcourir les composants et à agir en fonction de leur type. Passons sur l'ajout de texte et regardons ce qui se passe quand on rencontre un composant de début ou de fin de tableau. Dans le premier cas (début de tableau), on récupère dans le composant courant l'objet de la classe `tableau` instancié au moment de l'appel à la méthode `debutTable()` avec les paramètres appropriés. On place ce tableau dans la propriété `tableau` de l'objet, et on indique qu'on passe en mode table en affectant la valeur de la propriété `orientation`. À partir de là, cet objet devient disponible pour créer la mise en page des champs rencontrés ensuite.

Dans le second cas (fin de tableau), on vient de passer sur toutes les informations à placer dans le tableau et on peut donc produire la représentation HTML de ce dernier avec *tableauHTML()*, la concaténer au formulaire, et annuler le mode tableau en affectant la chaîne nullle à `orientation`.

C'est donc l'objet tableau qui se charge entièrement de la mise en forme des lignes et colonnes permettant d'aligner proprement les champs du formulaire. Bien entendu, il faut entre les composants de début et de fin de table alimenter le tableau avec ces champs : c'est ce que fait la partie traitant les composants de type `CHAMP`. Il suffit d'appeler la méthode *ajoutValeur()* de la classe `Tableau` en fonction de l'orientation souhaitée.

- En mode Table, vertical, les libellés sont dans la première colonne et les champs dans la seconde. On indexe les lignes par l'identifiant du composant, la première colonne par '`libelle`' et la seconde par '`champ`'.
- En mode Table, horizontal, les libellés sont dans les en-têtes de colonne, chaque champ forme une colonne. On indexe donc chaque colonne par l'identifiant du composant, et l'unique ligne par `ligne`. Notez qu'on a indiqué, au moment de l'instanciation du tableau, que cette ligne devait être répétée plusieurs fois.

Enfin, en mode libre (`orientation` est vide), on écrit simplement le libellé suivi du champ.

En résumé, la classe `Formulaire` se limite, du point de vue de l'application, à l'ensemble des méthodes présentées dans le tableau 3.8, page 154. En ce qui concerne la partie privée de la classe, si elle est bien conçue, il n'y aura plus à y revenir que ponctuellement pour quelques améliorations, comme par exemple ajouter des attributs HTML, gérer des classes CSS de présentation, introduire un système de contrôles JavaScript, etc. L'utilisation des objets produits par la classe est beaucoup plus simple que son implantation qui montre un exemple assez évolué de gestion interne d'une structure complexe, pilotée par une interface simple.

3.4 LA CLASSE IHMBD

Nous allons conclure par un dernier exemple de classe très représentatif d'un aspect important de la programmation objet, à savoir la capacité d'allier une réalisation *générique* (c'est-à-dire adaptée à toutes les situations) de tâches répétitives, et l'adaptation (voire le remplacement complet) de cette réalisation pour résoudre des cas particuliers. Le cas d'école considéré ici est celui des opérations effectuées avec PHP sur les tables d'une base de données. Les quelques chapitres qui précèdent ont montré que ces opérations sont souvent identiques dans leurs principes, mais varient dans le détail en fonction :

- de la structure particulière de la table ;
- de règles de gestion spécifiques comme, par exemple, la restriction à la liste des valeurs autorisées pour un attribut.

Les règles de gestion sont trop hétéroclites pour qu'on puisse les pré-réaliser simplement et en toute généralité. Leur codage au cas par cas semble inévitable. En revanche la structure de la table est connue et il est tout à fait envisageable d'automatiser les opérations courantes qui s'appuient sur cette structure, à savoir :

1. la recherche d'une ligne de la table par sa clé ;

2. la représentation de la table par un tableau HTML ;
3. la production d'un formulaire de saisie ou de mise à jour ;
4. des contrôles, avant toute mise à jour, sur le type ou la longueur des données à insérer ;
5. enfin la production d'une interface de consultation, saisie ou mise à jour semblable à celle que nous avons étudiée page 78.

La classe `IhmBD` (pour « Interface homme-machine et Bases de Données ») est une implantation de toutes ces fonctionnalités. Elle permet d'obtenir sans aucun effort, par simple instanciation d'un objet suivi d'un appel de méthode, une interface complète sur une table de la base. Bien entendu, cette interface peut s'avérer insatisfaisante du point de vue de l'ergonomie, de la présentation, ou du respect des règles particulières de gestion pour une table donnée. Dans ce cas on peut soit utiliser certaines méthodes pour régler des choix de présentation, soit définir une sous-classe spécialisée. Tous ces aspects sont développés dans ce qui suit.

Cette classe est un bon exemple du processus d'abstraction mis en œuvre couramment en programmation objet, et visant à spécifier de manière générale un comportement commun à de nombreuses situations (ici l'interaction avec une base de données). Le bénéfice de ce type de démarche est double. En premier lieu on obtient des outils pré-définis qui réduisent considérablement la réalisation d'applications. En second lieu on normalise l'implantation en décrivant à l'avance toutes les méthodes à fournir pour résoudre un problème donné. Tout cela aboutit à une économie importante d'efforts en développement et en maintenance. Dernier avantage : la description de la classe va nous permettre de récapituler tout ce que nous avons vu sur les techniques d'accès à MySQL (ou plus généralement à une base de données) avec PHP.

3.4.1 Utilisation

Dans sa version la plus simple, l'utilisation de la classe est élémentaire : on instancie un objet en indiquant sur quelle table on veut construire l'interface, on indique quelques attributs de présentation, et on appelle la méthode *genererIHM()*. Les quelques lignes de code qui suivent, appliquées à la table *Carte* qui a déjà servi pour la mini-application « Prise de commandes au restaurant » (voir page 99), suffisent.

Exemple 3.10 *exemples/AppIClasseIhmBD.php* : *Application de la classe* `ImhBD`.

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Création d'un formulaire</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
```

```
<body>

<?php
require_once("BDMySQL.php");
require_once("IhmBD.php");
require ("Normalisation.php");
require ("Connect.php");

// Normalisation des entrées HTTP
Normalisation();

try {
  // Connexion à la base
  $bd = new BDMySQL (NOM, PASSE, BASE, SERVEUR);

  // Creation de l'interface sur la table Carte
  $ihm = new IhmBD ("Carte", $bd);

  // Les en-têtes (pas obligatoire: le nom du champ sert d'en-
  // tête sinon)
  $ihm->setEntete("id_choix", "Numéro du plat");
  $ihm->setEntete("libelle", "Libellé du plat");
  $ihm->setEntete("type", "Type du plat");

  // Génération de l'interface
  echo $ihm->genererIHM($_REQUEST);
}
catch (Exception $exc) {
  echo "<b>Erreur rencontrée:</b> " . $exc->getMessage() . "\n";
}
?>
```

Bien entendu on réutilise la classe `BDMySQL` qui fournit tous les services nécessaires pour accéder à la base, de même que la classe `Tableau` nous servira pour les tableaux et la classe `Formulaire` pour les formulaires. Notez que l'utilisation d'une classe normalisée pour accéder à la base de données signifie que tout ce qui est décrit ci-dessous fonctionne également avec un SGBD autre que MySQL, en instanciant simplement un objet *bd* servant d'interface avec ce SGBD et conforme aux spécifications de la classe abstraite `BD` (voir page 130). La figure 3.7 montre l'affichage obtenu avec le script précédent. Il s'agit de bien plus qu'un affichage d'ailleurs : on peut insérer de nouvelles lignes, ou choisir de modifier l'une des lignes existantes à l'aide du formulaire. L'ajout de la fonction de destruction est, comme un certain nombre d'autres fonctionnalités, laissée en exercice au lecteur.

On obtient donc un outil en partie semblable à ce qu'offre *phpMyAdmin*. La structure de la table est récupérée de MySQL (ou de tout autre SGBD) et utilisée pour produire le formulaire, le tableau, les contrôles, etc. Bien entendu *phpMyAdmin* propose beaucoup plus de choses, mais il existe une différence de nature avec la classe `IhmBD`. Alors que *phpMyAdmin* est un outil intégré, nos objets fournissent des briques

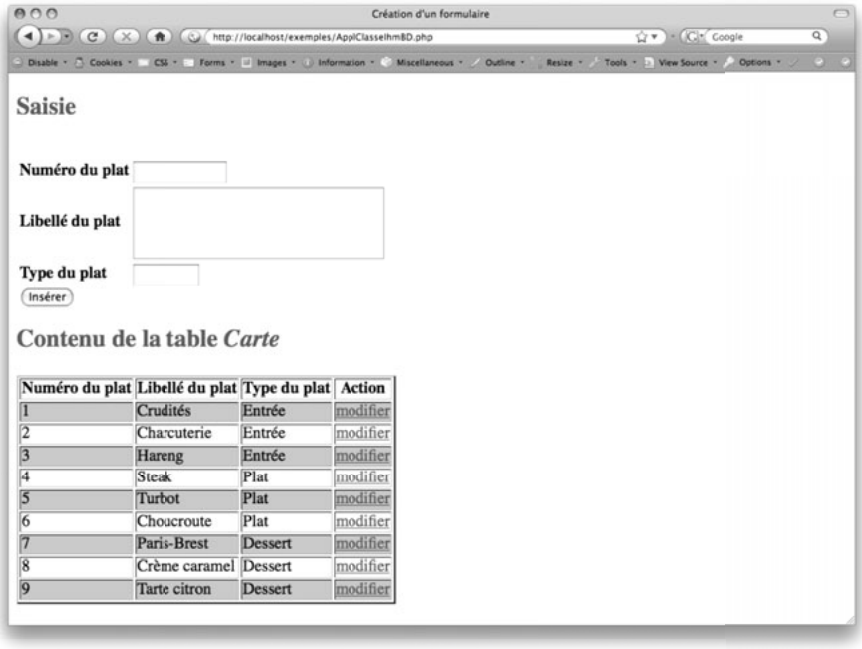

Création d'un formulaire
http://localhost/exemples/ApplClasseIhmBD.php
Saisie
Numéro du plat
Libellé du plat
Type du plat
Insérer
Contenu de la table Carte
Numéro du plat | Libellé du plat | Type du plat | Action
1 | Crudités | Entrée | modifier
2 | Charcuterie | Entrée | modifier
3 | Hareng | Entrée | modifier
4 | Steak | Plat | modifier
5 | Turbot | Plat | modifier
6 | Choucroute | Plat | modifier
7 | Paris-Brest | Dessert | modifier
8 | Crème caramel | Dessert | modifier
9 | Tarte citron | Dessert | modifier

Figure 3.7 — Affichage de l'interface sur la table *Carte*.

logicielles qui peuvent être intégrées dans toute application utilisant les méthodes publiques énumérées dans la table 3.9. Elles constituent une panoplie des accès à une table, à l'exception de l'ouverture d'un curseur pour accéder à un sous-ensemble des lignes.

Tableau 3.9 — Les méthodes publiques de la classe `IhmBD`

Méthode	Description
formulaire(`action`, `ligne`)	Renvoie un formulaire en saisie ou en mise à jour sur une ligne.
insertion(`ligne`)	Insère d'une ligne.
maj(`ligne`)	Met à jour d'une ligne.
tableau(`attributs`)	Renvoie un tableau HTML avec le contenu de la table.
setEntete(`nomAttribut`, `valeur`)	Affecte un en-tête descriptif à un attribut.
chercheLigne(`ligne`, `format`)	Renvoie une ligne recherchée par sa clé, au format tableau associatif ou objet.
genererIHM(`paramsHTTP`)	Produit une interface de consultation/mise à jour, basée sur les interactions HTTP.

REMARQUE – Ce besoin de disposer d'outils génériques pour manipuler les données d'une base relationnelle à partir d'un langage de programmation, sans avoir à toujours effectuer répétitivement les mêmes tâches, est tellement répandu qu'il a été « normalisé » sous le nom d'*Object-Relational Mapping* (ORM) et intégré aux *frameworks* de développement tel que celui

présenté dans le chapitre 9. La classe hmBD est cependant légèrement différente puisqu'elle permet de générer des séquences de consultation/saisie/mise à jour, ce que les outils d'ORM ne font généralement pas.

Ces méthodes peuvent être utilisées individuellement ou par l'intermédiaire des interactions définies dans la méthode *genererIHM()*. Elles peuvent aussi être redéfinies ou spécialisées. On aimerait bien par exemple disposer d'une liste déroulante pour le type de plat dans le formulaire de la table *Carte*. Il suffit alors de définir une sous-classe `IhmCarte` dans laquelle on ne ré-implante que la méthode *formulaire()*. Toutes les autres méthodes héritées de la super-classe, restent donc disponibles.

3.4.2 Implantation

Voyons maintenant l'implantation de la classe. Tout repose sur la connaissance du schéma de la table, telle qu'elle est fournie par la méthode *schemaTable* de la classe `BD`. Rappelons (voir page 132) que cette méthode renvoie un tableau associatif avec, pour chaque attribut de la table, la description de ses propriétés (type, longueur, participation à une clé primaire). Ce tableau a donc deux dimensions :

1. la première est le nom de l'attribut décrit ;
2. la seconde est la propriété, soit `type`, soit `longueur`, soit `cle_primaire`, soit enfin `not_null`.

Dans l'exemple de la table *Carte*, on trouvera dans le tableau décrivant le schéma un élément `['id_choix']['type']` avec la valeur `integer`, un élément `['id_choix']['cle_primaire']` avec la valeur `true`, etc.

REMARQUE – La classe ne fonctionne que pour des tables dotées d'une clé primaire, autrement dit d'un ou plusieurs attributs dont la valeur identifie une ligne de manière unique. La présence d'une clé primaire est de toute façon indispensable : voir le chapitre 4.

Voici le début de la classe. On énumère quelques constantes locales, puis des propriétés dont l'utilité sera détaillée ultérieurement, et enfin le constructeur de la classe.

```
class IhmBD
{
  // ——— Partie privée : les constantes et les variables
  const INS_BD = 1;
  const MAJ_BD = 2;
  const DEL_BD = 3;
  const EDITER = 4;

  protected $bd, $nomScript, $nomTable, $schemaTable, $entetes;

  // Le constructeur
  function __construct ($nomTable, $bd, $script="moi")
  {
```

```
    // Initialisation des variables privées
    $this->bd = $bd;
    $this->nomTable = $nomTable;
    if ($script == "moi")
      $this->nomScript = $_SERVER['PHP_SELF'];
    else
      $this->nomScript = $script;

    // Lecture du schéma de la table
    $this->schemaTable = $bd->schemaTable($nomTable);

    // Par défaut, les textes des attributs sont leurs noms
    foreach ($this->schemaTable as $nom => $options)
        $this->entetes[$nom] = $nom;
}
```

Le constructeur prend en entrée un nom de table, un objet de la classe `BD` (potentiellement également instance d'une sous-classe de `BD` : `BDMySQL`, `BDPostgreSQL`, `BDSQLite`, etc.) et le nom du script gérant l'interface avec la table. On commence par copier ces données dans les propriétés de l'objet pour les conserver durant toute sa durée de vie[5]. On recherche également le schéma de la table grâce à l'objet `bd`, et on le stocke dans la propriété `schemaTable`. Si la table n'existe pas, l'objet `bd` lèvera en principe une exception qu'on pourrait « attraper » ici.

> **REMARQUE –** On reçoit un objet, `bd`, passé par référence, alors que toutes les autres variables sont passées par valeur (comportement adopté depuis PHP 5). On stocke également une référence à cet objet avec l'instruction :
>
> ```
> $this->bd = $bd;
> ```
>
> L'opérateur d'affectation, pour les objets, n'effectue pas une *copie* comme pour tous les autres types de données, mais une référence. La variable `$this->bd` et la variable `$bd` référencent donc le *même* objet après l'affectation ci-dessus (voir page 61 pour la présentation des références). Il s'ensuit que deux codes indépendants vont travailler sur le même objet, ce qui peut parfois soulever des problèmes. Le script appelant a en effet instancié `$bd` et peut à bon droit estimer que l'objet lui appartient et qu'il peut en faire ce qu'il veut. Un objet de la classe `IhmBD` a lui aussi accès à cet objet et va le conserver durant toute sa durée de vie. Chacun peut effectuer des opérations incompatibles (par exemple fermer la connexion à la base) avec des résultats potentiellement dangereux. On pourrait effectuer une véritable copie de l'objet avec l'opérateur `clone` :
>
> ```
> $this->bd = clone $bd;
> ```
>
> On s'assure alors qu'il n'y aura pas de problème posé par le partage d'un même objet, le prix (modique) à payer étant l'utilisation d'un peu plus de mémoire, et une opération de copie.

5. Par défaut, on utilise le script courant, où l'objet aura été instancié, et dénoté `$_SERVER['PHP_SELF']`.

Voyons maintenant les principales méthodes de la classe `IhmBD`. La méthode *controle()* prend en entrée un tableau associatif contenant les données d'une ligne de la table manipulée. Elle doit être appelée avant une mise à jour. Son rôle est de contrôler autant que possible que tout va bien se passer au moment de l'exécution de la requête. Bien entendu une partie des contrôles dépend de règles spécifiques à la table manipulées qui ne peuvent pas être codées de manière générique, mais on peut toujours contrôler la longueur ou le type des données en fonction du schéma de la table. On peut aussi « échapper » les apostrophes avec la méthode *prepareChaine()* de la classe `BD`. C'est cette dernière manipulation qui est effectuée dans le code ci-dessous, le reste étant à compléter. Comme la plupart des méthodes données par la suite, *controle()* s'appuie sur le tableau `schema` pour connaître le nom des attributs de la table et y accéder.

```
// Méthode effectuant des contrôles avant mise à jour
protected function controle($ligne)
{
  $lignePropre = array();
  // On commence par traiter toutes les chaînes des attributs
  foreach ($this->schemaTable as $nom => $options) {
      // Traitement des apostrophes
      $lignePropre[$nom] = $this->bd->prepareChaine($ligne[$nom])
          ;
    }
  // On peut, de plus, contrôler le type ou la longueur des
  // données d'après le schéma de la table... A faire!

  return $lignePropre;
}
```

La méthode *controle()* prend un tableau en entrée, copié du script appelant vers l'espace des variables de la fonction, et renvoie un tableau en sortie, copié de la fonction vers le script appelant. Si on a peur que cela nuise aux performances, il reste toujours possible de recourir à un passage par référence.

La méthode *formulaire()* est donnée ci-dessous. Elle renvoie un formulaire adapté au mode de mise à jour à effectuer (insertion ou modification, voir page 78 pour les principes de création de ce type de formulaire).

```
  // Création d'un formulaire générique
  public function formulaire ($action, $ligne)
  {
    // Création de l'objet formulaire
    $form = new Formulaire ("post", $this->nomScript, false);

    $form->champCache ("action", $action);

    $form->debutTable();
    // Pour chaque attribut, création d'un champ de saisie
    foreach ($this->schemaTable as $nom => $options) {
        // D'abord vérifier que la valeur par défaut existe
```

```
            if (!isSet($ligne[$nom])) $ligne[$nom] = "";

            // Attention: traitement des balises HTML avant
            // affichage
            $ligne[$nom] = htmlSpecialChars($ligne[$nom]);

            // On met la clé primaire en champ caché
            if ($options['cle_primaire'] and $action == IhmBD::
                MAJ_BD) {
               $form->champCache ($nom, $ligne[$nom]);
            }
          else {
            // Affichage du champ
            if ($options['type'] == "blob")
              $form->champfenetre ($this->entetes[$nom],
                                   $nom, $ligne[$nom],
                                   4, 30);
            else
              $form->champTexte ($this->entetes[$nom],
                                 $nom, $ligne[$nom],
                                 $options['longueur']);
            }
      }
      $form->finTable();

      if ($action == IhmBD::MAJ_BD)
        $form->champValider ("Modifier", "submit");
      else
        $form->champValider ("Insérer", "submit");

      return $form->formulaireHTML();
    }
```

Noter l'utilisation de la fonction `htmlSpecialChars()` pour traiter les données venant de la base afin d'éviter les inconvénients résultant de la présence de balises HTML dans ces données (sujet traité page 64). La méthode utilise bien entendu la classe `Formulaire` pour aligner régulièrement chaque champ avec son en-tête. De même, la méthode `tableau()` ci-dessous s'appuie sur un objet de la classe `Tableau`. Là aussi on prend soin d'appliquer `htmlSpecialChars()` aux données provenant de la base.

```
    // Création d'un tableau générique
  public function tableau($attributs=array())
  {
    // Création de l'objet Tableau
    $tableau = new Tableau(2, $attributs);
    $tableau->setCouleurImpaire("silver");
    $tableau->setAfficheEntete(1, false);

    // Texte des en-têtes
    foreach ($this->schemaTable as $nom => $options)
    $tableau->ajoutEntete(2, $nom, $this->entetes[$nom]);
```

```
        $tableau->ajoutEntete(2, "action", "Action");

        // Parcours de la table
        $requete = "SELECT * FROM $this->nomTable";
        $resultat = $this->bd->execRequete ($requete);

        $i=0;
        while ($ligne = $this->bd->ligneSuivante ($resultat)) {
         $i++;
         // Création des cellules
         foreach ($this->schemaTable as $nom => $options)  {
           // Attention: traitement des balises HTML avant affichage
           $ligne[$nom] = htmlSpecialChars($ligne[$nom]);
           $tableau->ajoutValeur($i, $nom, $ligne[$nom]);
         }

         // Création de l'URL de modification
         $urlMod = $this->accesCle($ligne) . "&amp;action=" . IhmBD::
             EDITER;
         $modLink = "<a href='$this->nomScript?$urlMod'>modifier</a>";

         $tableau->ajoutValeur($i, "action", $modLink);
       }

        // Retour de la chaîne contenant le tableau
        return $tableau->tableauHTML();
      }
```

Je laisse le lecteur consulter directement le code des méthodes `accesCle()`, `insertion()`, `maj()` et `chercheLigne()` qui construisent simplement des requêtes SQL `SELECT`, `INSERT` ou `UPDATE` en fonction du schéma de la table et des données passées en paramètre. La dernière méthode intéressante est `genererIHM()` qui définit les interactions avec le formulaire et le tableau. Trois actions sont possibles :

1. on a utilisé le formulaire pour effectuer une *insertion* : dans ce cas on exécute la méthode `insertion()` avec les données reçues par HTTP ;
2. on a utilisé le formulaire pour effectuer une *mise à jour* : dans ce cas on exécute la méthode `maj()` ;
3. on a utilisé l'ancre *modifier* du tableau pour éditer une ligne et la modifier : dans ce cas on appelle le formulaire en mise à jour.

Si le formulaire n'est pas utilisé en mise à jour, on l'affiche en mode insertion. Dans tous les cas, on affiche le tableau contenant les lignes, ce qui donne le code suivant :

```
public function genererIHM ($paramsHTTP)
{
  // A-t'on demandé une action?
  if (isSet($paramsHTTP['action']))
  $action = $paramsHTTP['action'];
  else
  $action = "";

  $affichage = "";
  switch ($action) {
    case IhmBD::INS_BD:
      // On a demandé une insertion
      $this->insertion($paramsHTTP);
      $affichage .= "<i>Insertion effectuée.</i>";
      break;

    case IhmBD::MAJ_BD:
      // On a demandé une modification
      $this->maj($paramsHTTP);
      $affichage .= "<i>Mise à jour effectuée.</i>";
      break;

    case IhmBD::EDITER:
      // On a demandé l'accès à une ligne en mise à jour
      $ligne  = $this->chercheLigne ($paramsHTTP);
      $affichage .= $this->formulaire(IhmBD::MAJ_BD, $ligne);
      break;
  }

  // Affichage du formulaire en insertion si on n'a pas édité
  // en mise à jour
  if ($action != IhmBD::EDITER) {
    $affichage .= "<h2>Saisie </h2>";
    $affichage .= $this->formulaire(IhmBD::INS_BD, array());
  }

  // On met toujours le tableau du contenu de la table
  $affichage .= "<h2>Contenu de la table <i>$this->nomTable</i>
      </h2>"
  . $this->tableau(array("border" => 2));
  // Retour de la page HTML
  return $affichage;
}
```

Cette classe fournit ainsi une version « par défaut » des fonctionnalités d'accès à une table, version qui peut suffire pour élaborer rapidement une interface. Pour des besoins plus sophistiqués, il est possible de spécialiser cette classe pour l'adapter aux contraintes et règles de manipulation d'une table particulière. Le chapitre 5 donne un exemple complet d'une telle spécialisation (voir page 267). À titre de mise en bouche, voici la sous-classe `IhmCarte` qui surcharge la méthode `formulaire()` pour présenter les types de plat sous la forme d'une liste déroulante.

Exemple 3.11 *exemples/IhmCarte.php* : *La sous-classe* `IhmCarte`

```
<?
require_once("IhmBD.class.php");

// Classe étendant IhmBD, spécialisée pour la table Carte

class IhmCarte extends IhmBD
{
  // La carte est caractérisée par les types de plats autorisés
  private $typesAutorises = array("EntrÈe" => "EntrÈe",
                                  "Plat" => "Plat",
                                  "Dessert" => "Dessert");

  // Le constructeur de la classe. Attention à bien penser
  // à appeler le constructeur de la super-classe.

  function __construct($nomTable, $bd, $script="moi")
  {
    // Appel du constructeur de IhmBD
    parent::__construct($nomTable, $bd, $script);

    // On peut placer les entÍtes dËs maintenant
    $this->entetes['id_choix'] = "NumÈro du plat";
    $this->entetes['libelle'] = "LibellÈ du plat";
    $this->entetes['type'] = "Type du plat";
  }

  /*****************    Partie publique ********************/

  // Redéfinition du formulaire
  public function formulaire ($action, $ligne)
  {
    // Création de l'objet formulaire
    $form = new Formulaire ("get", $this->nomScript, false);

    $form->champCache ("action", $action);
    $form->debutTable();

    // En mise à jour, la clé est cachée, sinon elle est
        saisissable
    if ($action == IhmBD::MAJ_BD)
    $form->champCache ("id_choix",  $ligne['id_choix']);
    else
    $form->champTexte($this->entetes['id_choix'], 'id_choix', "",
         4);

    // Vérifier que la valeur par défaut existe
    if (!isSet($ligne['libelle'])) $ligne['libelle'] = "";
    if (!isSet($ligne['type'])) $ligne['type'] = "Entrée";

    $form->champTexte ($this->entetes['libelle'], "libelle",
```

```
        $ligne['libelle'], 30);
        $form->champListe ($this->entetes['type'], "type",
        $ligne['type'], 1,
        $this->typesAutorises);
        $form->finTable();

        if ($action == IhmBD::MAJ_BD)
        $form->champValider ("Modifier", "submit");
        else
        $form->champValider ("Insérer", "submit");

        return $form->formulaireHTML();
    }
}
?>
```

Seules deux méthodes sont surchargées : le constructeur et le formulaire. Pour le constructeur, notez que l'on combine un appel au constructeur de la classe générique avec `parent::__construct` et l'ajout de quelques initialisations. Quand on manipulera un objet de la classe `IhmCarte`, le constructeur et le formulaire seront ceux de la sous-classe, toutes les autres méthodes provenant par héritage de la super-classe.

DEUXIÈME PARTIE

Conception et création d'un site

À partir d'ici nous commençons la conception et la réalisation du site WEBSCOPE, une application complète de gestion d'une base de films et d'appréciations sur ces films. Comme pour les exemples, récupérez le code sur le site du livre et décompressez-le dans *htdocs*. La structure des répertoires est plus complexe que celle utilisée jusqu'à présent. Pour vous aider à retrouver les fichiers, les exemples du livre donnent leur nom en le préfixant par le chemin d'accès à partir de la racine `webscope`.

Vous trouverez dans le répertoire WEBSCOPE un fichier *LISEZ_MOI* qui indique comment installer l'application, créer la base et l'initialiser avec un ensemble de films et d'artistes.

4

Création d'une base MySQL

Ce chapitre présente le processus de conception et de définition du *schéma* d'une base MySQL. Le schéma correspond à tout ce qui relève de la description de la base. Il définit la forme de la base, ainsi que les contraintes que doit respecter son contenu.

La conception d'un schéma correct est essentielle pour le développement d'une application viable. Dans la mesure où la base de données est le fondement de tout le système, une erreur pendant sa conception est difficilement récupérable par la suite. Nous décrivons dans ce chapitre les principes essentiels, en mettant l'accent sur les pièges à éviter, ainsi que sur la méthode permettant de créer une base saine.

4.1 CONCEPTION DE LA BASE

La méthode généralement employée pour la conception de bases de données est de construire un *schéma Entité/Association* (E/A). Ces schémas ont pour caractéristiques d'être simples et suffisamment puissants pour représenter des bases relationnelles. De plus, la représentation graphique facilite considérablement la compréhension.

La méthode distingue les *entités* qui constituent la base de données, et les *associations* entre ces entités. Ces concepts permettent de donner une structure à la base, ce qui s'avère indispensable. Nous commençons par montrer les problèmes qui surviennent si on traite une base relationnelle comme un simple fichier texte, ce que nous avons d'ailleurs fait, à peu de choses près, jusqu'à présent.

4.1.1 Bons et mauvais schémas

Reprenons la table *FilmSimple* largement utilisée dans les chapitres précédents. Voici une représentation de cette table, avec le petit ensemble de films sur lequel nous avons travaillé.

titre	année	nom_realisateur	prénom_realisateur	annéeNaiss
Alien	1979	Scott	Ridley	1943
Vertigo	1958	Hitchcock	Alfred	1899
Psychose	1960	Hitchcock	Alfred	1899
Kagemusha	1980	Kurosawa	Akira	1910
Volte-face	1997	Woo	John	1946
Pulp Fiction	1995	Tarantino	Quentin	1963
Titanic	1997	Cameron	James	1954
Sacrifice	1986	Tarkovski	Andrei	1932

L'objectif de cette table est clair. Il s'agit de représenter des films avec leur metteur en scène. Malheureusement, même pour une information aussi simple, il est facile d'énumérer tout un ensemble de problèmes potentiels. Tous ou presque découlent d'un grave défaut de la table ci-dessus : il est possible de représenter la même information plusieurs fois.

Anomalies lors d'une insertion

Rien n'empêche de représenter plusieurs fois le même film. Pire : il est possible d'insérer plusieurs fois le film *Vertigo* en le décrivant à chaque fois de manière différente, par exemple en lui attribuant une fois comme réalisateur Alfred Hitchcock, puis une autre fois John Woo, etc.

Une bonne question consiste d'ailleurs à se demander ce qui distingue deux films l'un de l'autre, et à quel moment on peut dire que la même information a été répétée. Peut-il y avoir deux films différents avec le même titre par exemple ? Si la réponse est « non », alors on devrait pouvoir assurer qu'il n'y a pas deux lignes dans la table avec la même valeur pour l'attribut `titre`. Si la réponse est « oui », il reste à déterminer quel est l'ensemble des attributs qui permet de caractériser de manière unique un film.

Anomalies lors d'une modification

La redondance d'informations entraîne également des anomalies de mise à jour. Supposons que l'on modifie l'année de naissance de Hitchcock pour la ligne *Vertigo* et pas pour la ligne *Psychose*. On se retrouve alors avec des informations incohérentes.

Les mêmes questions que précédemment se posent d'ailleurs. Jusqu'à quel point peut-on dire qu'il n'y a qu'un seul réalisateur nommé Hitchcock, et qu'il ne doit donc y avoir qu'une seule année de naissance pour un réalisateur de ce nom ?

Anomalies lors d'une destruction

On ne peut pas supprimer un film sans supprimer du même coup son metteur en scène. Si on souhaite, par exemple, ne plus voir le film *Titanic* figurer dans la base de données, on va effacer du même coup les informations sur James Cameron.

La bonne méthode

Une bonne méthode évitant les anomalies ci-dessus consiste à ;

1. être capable de représenter individuellement les films et les réalisateurs, de manière à ce qu'une action sur l'un n'entraîne pas systématiquement une action sur l'autre ;
2. définir une méthode *d'identification* d'un film ou d'un réalisateur, qui permette d'assurer que la même information est représentée une seule fois ;
3. préserver le lien entre les films et les réalisateurs, mais sans introduire de redondance.

Commençons par les deux premières étapes. On va distinguer la table des films et la table des réalisateurs. Ensuite, on décide que deux films ne peuvent avoir le même titre, mais que deux réalisateurs peuvent avoir le même nom. Afin d'avoir un moyen d'identifier les réalisateurs, on va leur attribuer un numéro, désigné par `id`. On obtient le résultat suivant, les identifiants (ou *clés*) étant en gras.

Tableau 4.1 — La table des films

titre	année
Alien	1979
Vertigo	1958
Psychose	1960
Kagemusha	1980
Volte-face	1997
Pulp Fiction	1995
Titanic	1997
Sacrifice	1986

Tableau 4.2 — La table des réalisateurs

id	nom_realisateur	prénom_realisateur	année_naissance
1	Scott	Ridley	1943
2	Hitchcock	Alfred	1899
3	Kurosawa	Akira	1910
4	Woo	John	1946
5	Tarantino	Quentin	1963
6	Cameron	James	1954
7	Tarkovski	Andrei	1932

Premier progrès : il n'y a maintenant plus de redondance dans la base de données. Le réalisateur Hitchcock, par exemple, n'apparaît plus qu'une seule fois, ce qui élimine les anomalies de mise à jour évoquées précédemment.

Il reste à représenter le lien entre les films et les metteurs en scène, sans introduire de redondance. Maintenant que nous avons défini les identifiants, il existe un moyen simple pour indiquer quel est le metteur en scène qui a réalisé un film : associer

l'identifiant du metteur en scène au film. On ajoute un attribut `id_realisateur` dans la table *Film*, et on obtient la représentation suivante.

Tableau 4.3 — La table des films

titre	année	*id_realisateur*
Alien	1979	1
Vertigo	1958	2
Psychose	1960	2
Kagemusha	1980	3
Volte-face	1997	4
Pulp Fiction	1995	5
Titanic	1997	6
Sacrifice	1986	7

Tableau 4.4 — La table des réalisateurs

id	nom_realisateur	prénom_realisateur	année_naissance
1	Scott	Ridley	1943
2	Hitchcock	Alfred	1899
3	Kurosawa	Akira	1910
4	Woo	John	1946
5	Tarantino	Quentin	1963
6	Cameron	James	1954
7	Tarkovski	Andrei	1932

Cette représentation est correcte. La redondance est réduite au minimum puisque, seule la clé identifiant un metteur en scène a été déplacée dans une autre table (on parle de *clé étrangère*). On peut vérifier que toutes les anomalies citées ont disparu.

Anomalie d'insertion. Maintenant que l'on sait quelles sont les caractéristiques qui identifient un film, il est possible de déterminer au moment d'une insertion si elle va introduire ou non une redondance. Si c'est le cas, on doit interdire cette insertion.

Anomalie de mise à jour. Il n'y a plus de redondance, donc toute mise à jour affecte l'unique instance de la donnée à modifier.

Anomalie de destruction. On peut détruire un film sans affecter les informations sur le réalisateur.

Ce gain dans la qualité du schéma n'a pas pour contrepartie une perte d'information. Il est facile de voir que l'information initiale (autrement dit, avant décomposition) peut être reconstituée intégralement. En prenant un film, on obtient l'identité

de son metteur en scène, et cette identité permet de trouver *l'unique* ligne dans la table des réalisateurs qui contient toutes les informations sur ce metteur en scène. Ce processus de reconstruction de l'information, dispersée dans plusieurs tables, peut s'exprimer avec SQL.

La modélisation avec un graphique Entité/Association offre une méthode simple pour arriver au résultat ci-dessus, et ce même dans des cas beaucoup plus complexes.

4.1.2 Principes généraux

Un schéma E/A décrit l'application visée, c'est-à-dire une *abstraction* d'un domaine d'étude, pertinente relativement aux objectifs visés. Rappelons qu'une abstraction consiste à choisir certains aspects de la réalité perçue (et donc à éliminer les autres). Cette sélection se fait en fonction de certains *besoins* qui doivent être précisément analysés et définis.

Par exemple, pour le site *Films*, on n'a pas besoin de stocker dans la base de données l'intégralité des informations relatives à un internaute, ou à un film. Seules comptent celles qui sont importantes pour l'application. Voici le schéma décrivant la base de données du site *Films* (figure 4.1). Sans entrer dans les détails pour l'instant, on distingue

1. des *entités*, représentées par des rectangles, ici *Film*, *Artiste*, *Internaute* et *Pays* ;
2. des *associations entre entités* représentées par des liens entre ces rectangles. Ici on a représenté par exemple le fait qu'un artiste *joue* dans des films, qu'un internaute *note* des films, etc.

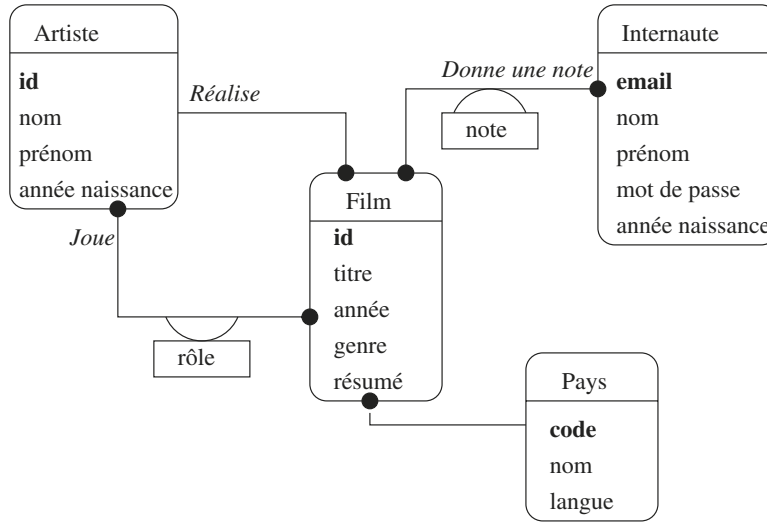


Figure 4.1 — Schéma de la base de données *Films*

Chaque entité est caractérisée par un ensemble d'attributs, parmi lesquels un ou plusieurs forment l'identifiant unique (en gras). Comme nous l'avons exposé

précédemment, il est essentiel de dire ce qui caractérise de manière unique une entité, de manière à éviter la redondance d'information.

Les associations sont caractérisées par des *cardinalités*. Le point noir sur le lien *Réalise*, du côté de l'entité *Film*, signifie qu'un artiste peut réaliser plusieurs films. L'absence de point noir du côté *Artiste* signifie en revanche qu'un film ne peut être réalisé que par un seul artiste. En revanche dans l'association *Donne une note*, un internaute peut noter plusieurs films, et un film peut être noté par plusieurs internautes, ce qui justifie la présence d'un • aux deux extrémités de l'association.

Le choix des cardinalités est *essentiel*. Ce choix est parfois discutable, et constitue, avec le choix des identifiants, l'aspect le plus délicat de la modélisation. Reprenons l'exemple de l'association *Réalise*. En indiquant qu'un film est réalisé par *un seul* metteur en scène, on s'interdit les – rares – situations où un film est réalisé par plusieurs personnes. Il ne sera donc pas possible de représenter dans la base de données une telle situation. Tout est ici question de choix et de compromis : est-on prêt en l'occurrence à accepter une structure plus complexe (avec un • de chaque côté) pour l'association *Réalise*, pour prendre en compte un nombre minime de cas ?

Outre les propriétés déjà évoquées (simplicité, clarté de lecture), évidentes sur ce schéma, on peut noter aussi que la modélisation conceptuelle est totalement indépendante de tout choix d'implantation. Le schéma de la figure 4.1 ne spécifie aucun système en particulier. Il n'est pas non plus question de type ou de structure de données, d'algorithme, de langage, etc. En principe, il s'agit donc de la partie la plus stable d'une application. Le fait de se débarrasser à ce stade de la plupart des considérations techniques permet de se concentrer sur l'essentiel : que veut-on stocker dans la base ?

Une des principales difficultés dans le maniement des schémas E/A est que la qualité du résultat ne peut s'évaluer que par rapport à une demande souvent floue et incomplète. Il est donc souvent difficile de valider (en fonction de quels critères ?) le résultat. Peut-on affirmer par exemple que :

1. que *toutes* les informations nécessaires sont représentées ?
2. qu'un film ne sera *jamais* réalisé par plus d'un artiste ?

Il faut faire des choix, en connaissance de cause, en sachant toutefois qu'il est toujours possible de faire évoluer une base de données, quand cette évolution n'implique pas de restructuration trop importante. Pour reprendre les exemples ci-dessus, il est facile d'ajouter des informations pour décrire un film ou un internaute ; il serait beaucoup plus difficile de modifier la base pour qu'un film passe de un, et un seul, réalisateur, à plusieurs. Quant à changer la clé de *Internaute*, c'est une des évolutions les plus complexes à réaliser. Les cardinalités et le choix des clés font vraiment partie des aspects décisifs des choix de conception.

4.1.3 Création d'un schéma E/A

Le modèle E/A, conçu en 1976, est à la base de la plupart des méthodes de conception. La syntaxe employée ici est celle de la méthode OMT, transcrite pratiquement à l'identique dans UML. Il existe beaucoup d'autres variantes, dont celle de la méthode MERISE principalement utilisée en France. Ces variantes sont globalement équivalentes. Dans tous les cas la conception repose sur deux concepts complémentaires, *entité* et *association*.

Entités

On désigne par *entité* tout objet ou concept *identifiable* et *pertinente* pour l'application. Comme nous l'avons vu précédemment, la notion *d'identité* est primordiale. C'est elle qui permet de distinguer les entités les unes des autres, et de dire qu'une information est redondante ou qu'elle ne l'est pas. Il est indispensable de prévoir un moyen technique pour pouvoir effectuer cette distinction entre entités au niveau de la base de données : on parle *d'identifiant* ou de *clé*. La pertinence est également essentielle : on ne doit prendre en compte que les informations nécessaires pour satisfaire les besoins. Par exemple :

1. le film *Impitoyable* ;
2. l'acteur *Clint Eastwood* ;

sont des entités pour la base *Films*.

La première étape d'une conception consiste à identifier les entités utiles. On peut souvent le faire en considérant quelques cas particuliers. La deuxième est de regrouper les entités en ensembles : en général, on ne s'intéresse pas à un individu particulier mais à des groupes. Par exemple il est clair que les films et les acteurs sont des ensembles distincts d'entités. Qu'en est-il de l'ensemble des réalisateurs et de l'ensemble des acteurs ? Doit-on les distinguer ou les assembler ? Il est certainement préférable de les assembler, puisque des acteurs peuvent aussi être réalisateurs. Chaque ensemble est désigné par son nom.

Les entités sont caractérisées par des propriétés : le nom, la date de naissance, l'adresse, etc. Ces propriétés sont dénotées *attributs* dans la terminologie du modèle E/A. Il n'est pas question de donner exhaustivement toutes les caractéristiques d'une entité. On ne garde que celles utiles pour l'application.

Par exemple le titre et l'année de production sont des attributs des entités de la classe *Film*. Il est maintenant possible de décrire un peu plus précisément le contenu d'un ensemble d'entités par un *type* qui est constitué des éléments suivants :

1. son nom (par exemple, *Film*) ;
2. la liste de ses attributs ;
3. l'indication du (ou des) attribut(s) permettant d'identifier l'entité : ils constituent la *clé*.

Un type d'entité, comprenant les éléments ci-dessus, décrit toutes les entités d'un même ensemble. On représente ce type graphiquement comme sur la figure 4.2 qui donne l'exemple de deux entités, *Internaute* et *Film*.

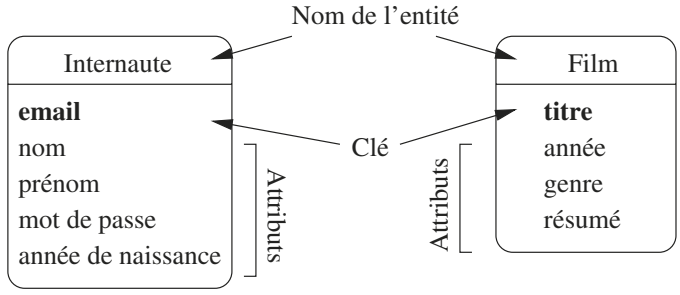


Figure 4.2 — Représentation des entités

Choix de l'identifiant

Dans le premier cas, on a décidé qu'un internaute était identifié par son email, ce qui est cohérent pour une application web. Il est en fait très rare de trouver un attribut d'une entité pouvant jouer le rôle d'identifiant. Le choix du titre pour identifier un film serait par exemple beaucoup plus discutable. En ce qui concerne les artistes, acteurs ou réalisateurs, l'identification par le nom seul paraît vraiment impossible. On pourrait penser à utiliser la paire `(nom,prénom)`, mais l'utilisation d'identifiants composés de plusieurs attributs, quoique possible, peut poser des problèmes de performance et complique les manipulations par SQL.

Dans la situation, fréquente, où on a du mal à déterminer quelle est la clé d'une entité, on crée un identifiant « abstrait » indépendant de tout autre attribut. Pour les artistes, nous avons ajouté **id**, un numéro séquentiel qui sera incrémenté au fur et à mesure des insertions. Ce choix est souvent le meilleur, dès lors qu'un attribut ne s'impose pas de manière évidente comme clé.

Associations

La représentation (et le stockage) d'entités indépendantes les unes des autres est de peu d'utilité. On va maintenant décrire les *associations* entre des ensembles d'entités. Une bonne manière de comprendre comment on doit représenter une association entre des ensembles d'entités est de faire un graphe illustrant quelques exemples, les plus généraux possibles.

Prenons le cas de l'association représentant le fait qu'un réalisateur met en scène des films. Sur le graphe de la figure 4.3 on remarque que :

1. certains réalisateurs mettent en scène plusieurs films ;
2. inversement, un film est mis en scène par au plus un réalisateur.

La recherche des situations les plus générales possibles vise à s'assurer que les deux caractéristiques ci-dessus sont vraies dans tout les cas. Bien entendu on peut trouver

1% des cas où un film a plusieurs réalisateurs, mais la question se pose alors : doit-on modifier la structure de notre base, pour 1% des cas. Ici, on a décidé que non.

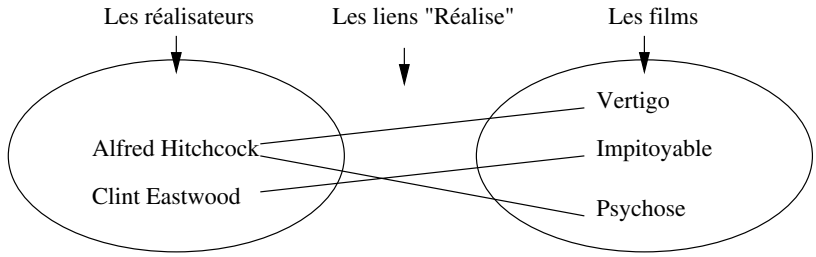


Figure 4.3 — Association entre deux ensembles.

Ces caractéristiques sont essentielles dans la description d'une association. On les représente sur le schéma de la manière suivante :

1. si une entité *A* peut être liée à plusieurs entités *B*, on indique cette multiplicité par un point noir (•) à l'extrémité du lien allant de *A* vers *B* ;
2. si une entité *A* peut être liée à au plus une entité *B*, alors on indique cette unicité par un trait simple à l'extrémité du lien allant de *A* vers *B* ;

Pour l'association entre *Réalisateur* et *Film*, cela donne le schéma de la figure 4.4. Cette association se lit *Un réalisateur réalise un film*, mais on pourrait tout aussi bien utiliser la forme passive avec comme intitulé de l'association *Est réalisé par* et une lecture *Un film est réalisé par un réalisateur*. Le seul critère à privilégier dans ce choix des termes est la clarté de la représentation.

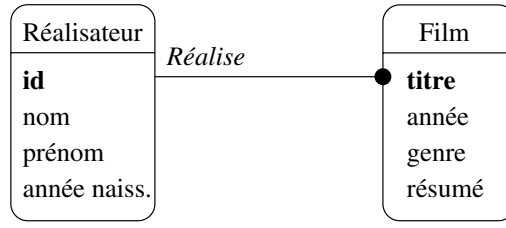


Figure 4.4 — Représentation de l'association.

Prenons maintenant l'exemple de l'association `(Acteur,Film)` représentant le fait qu'un acteur joue dans un film. Un graphe basé sur quelques exemples est donné dans la figure 4.5. On constate tout d'abord qu'un acteur peut jouer dans plusieurs films, et que dans un film on trouve plusieurs acteurs. Mieux : Clint Eastwood, qui apparaissait déjà en tant que metteur en scène, est maintenant également acteur, et dans le même film.

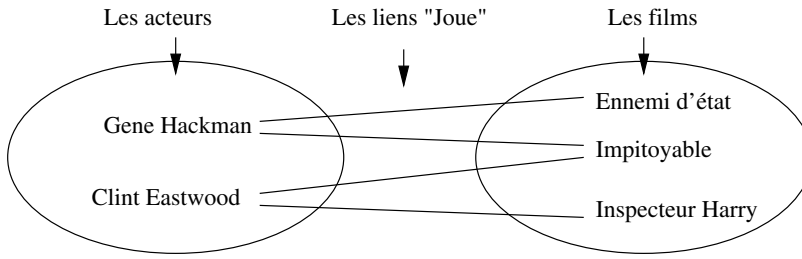


Figure 4.5 — Association `(Acteur,Film)`

Cette dernière constatation mène à la conclusion qu'il vaut mieux regrouper les acteurs et les réalisateurs dans un même ensemble, désigné par le terme plus général « Artiste ». On obtient le schéma de la figure 4.6, avec les deux associations représentant les deux types de lien possible entre un artiste et un film : il peut jouer dans le film, ou le réaliser. Ce « ou » n'est pas exclusif : Eastwood joue dans *Impitoyable*, qu'il a aussi réalisé.

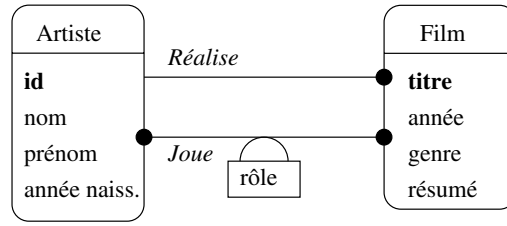


Figure 4.6 — Associations entre *Artiste* et *Film.*

Dans le cas d'associations avec des cardinalités multiples de chaque côté, certains attributs doivent être affectés qu'à l'association elle-même. Par exemple, l'association *Joue* a pour attribut le rôle tenu par l'acteur dans le film (figure 4.6). Clairement, on ne peut associer cet attribut ni à *Acteur* puisqu'il a autant de valeurs possibles qu'il y a de films dans lesquels cet acteur a joué, ni à *Film*, la réciproque étant vraie également. Seules les associations ayant des cardinalités multiples de chaque côté peuvent porter des attributs.

Associations impliquant plus de deux entités

On peut envisager des associations entre plus de deux entités, mais elles sont plus difficiles à comprendre, et la signification des cardinalités devient beaucoup plus ambiguë.

Imaginons que l'on souhaite représenter dans la base de données les informations indiquant que tel film passe dans telle salle de cinéma à tel horaire. On peut être tenté de représenter cette information en ajoutant des entités *Salle* et *Horaire*, et en créant une association ternaire comme celle de la figure 4.7.

Avec un peu de réflexion, on décide que plusieurs films peuvent passer au même horaire (mais pas dans la même salle !), qu'un film est vu à plusieurs horaires différents, et que plusieurs films passent dans la même salle (mais pas au même horaire !). D'où les cardinalités multiples pour chaque lien. On peut affecter des attributs à cette association, comme par exemple le tarif, qui dépend à la fois de l'horaire, du film et de la salle.

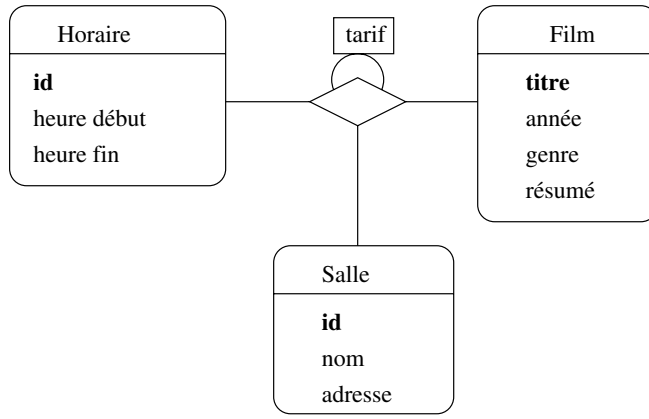


Figure 4.7 — Association ternaire.

Ces associations avec plusieurs entités sont assez difficiles à interpréter, et elle offrent beaucoup de liberté sur la représentation des données, ce qui n'est pas toujours souhaitable. On ne sait pas par exemple interdire que deux films passent dans la même salle au même horaire. Le graphe de la figure 4.8 montre que cette configuration est tout à fait autorisée.

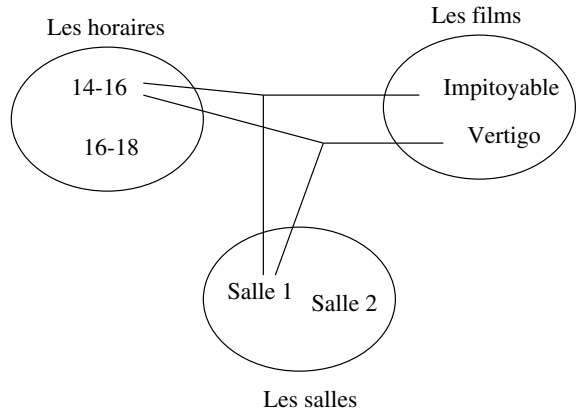


Figure 4.8 — Graphe d'une association ternaire.

Les associations autres que binaires sont donc à éviter dans la mesure du possible. Il est *toujours* possible de remplacer une telle association par une entité. Sur l'exemple précédent, on peut tout simplement remplacer l'association

ternaire par une entité *Séance*, qui est liée, par des associations binaires, aux trois entités existantes (voir figure 4.9). Il est préférable d'avoir plus d'entités et moins d'associations complexes, car cela rend l'interprétation du schéma plus facile. Dans le cas de la séance, au lieu de considérer simultanément trois entités et une association, on peut prendre maintenant séparément trois paires d'entités, chaque paire étant liée par une association binaire.

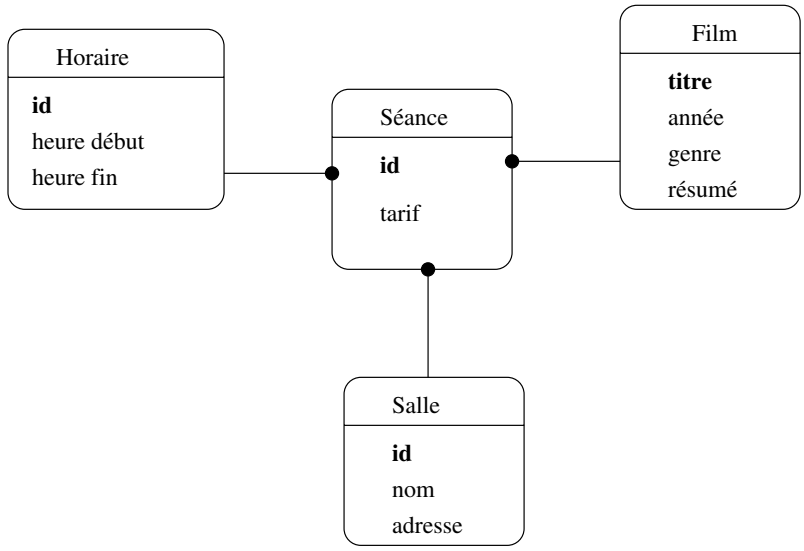


Figure 4.9 — Transformation d'une association ternaire en entité.

Récapitulatif

En résumé, la méthode basée sur les graphiques Entité/Association est simple et pratique. Elle n'est malheureusement pas infaillible, et repose sur une certaine expérience. Le principal inconvénient est qu'il n'y a pas de règle absolue pour déterminer ce qui est entité, attribut ou association, comme le montre l'exemple précédent où une association s'est transformée en entité.

À chaque moment de la conception d'une base, il faut se poser des questions auxquelles on répond en se basant sur quelques principes de bon sens :

1. rester le plus simple possible ;
2. éviter les associations compliquées ;
3. ne pas représenter plusieurs fois la même chose ;
4. ne pas mélanger dans une même entité des concepts différents.

Voici quelques exemples de questions légitimes, et de réponses qui paraissent raisonnables.

« Est-il préférable de représenter le metteur en scène comme un attribut de Film ou comme une association avec Artiste ? »

Réponse : comme une association, car on connaît alors non seulement le nom, mais aussi toutes les autres propriétés (prénom, année de naissance, ...). De plus, ces informations peuvent être associées à beaucoup d'autres films. En utilisant une association, on permet à tous ces films de référencer le metteur en scène, en évitant la répétition de la même information à plusieurs endroits.

« Est-il indispensable de gérer une entité Horaire ? »

Réponse : pas forcément ! D'un côté, cela permet de normaliser les horaires. Plusieurs séances peuvent alors faire référence au même horaire, avec les avantages en termes de gain de place et de cohérence cités précédemment. En revanche, on peut considérer que cela alourdit le schéma inutilement. On peut alors envisager de déplacer les attributs de *Horaire* (soit `heure début` et `heure fin`) dans *Séance*.

« Pourquoi ne pas mettre le nom du pays comme attribut de Film ? »

C'est envisageable. Mais l'utilité d'associer un film au pays qui l'a produit est certainement de pouvoir faire des classements par la suite. Il s'agit d'une situation typique où on utilise une *codification* pour ranger des données par catégorie. Si on met le nom du pays comme attribut, l'utilisateur peut saisir librement une chaîne de caractères quelconque, et on se retrouve avec « U.S.A », «États Unis », « U.S », etc, pour désigner les États-Unis, ce qui empêche tout regroupement par la suite. Le fait de référencer une codification imposée, représentée dans *Pays*, force les valeurs possibles, en les normalisant.

4.2 SCHÉMA DE LA BASE DE DONNÉES

La création d'un schéma MySQL est simple une fois que l'on a déterminé les entités et les associations. En pratique, on transcrit le schéma E/A en un schéma relationnel comprenant toutes les tables de la base, en suivant quelques règles données dans ce qui suit. Nous prenons bien entendu comme exemple le schéma de la base *Film*, tel qu'il est donné dans la figure 4.1, page 185.

4.2.1 Transcription des entités

On passe donc d'un modèle disposant de deux structures (entités et associations) à un modèle disposant d'une seule (tables). Logiquement, entités et associations seront toutes deux transformées en tables. La subtilité réside dans la nécessité de préserver les *liens* existants dans un schéma E/A et qui semblent manquer dans les schémas de tables. Dans ce dernier cas, on utilise un mécanisme de référence par valeur basé sur les *clés* des tables.

La clé d'une table est le plus petit sous-ensemble des attributs qui permet d'identifier chaque ligne de manière unique. Nous avons omis de spécifier la clé dans certaines tables des chapitres précédents, ce qui doit absolument être évité

quand on passe à une application sérieuse. Une table doit *toujours* avoir une clé. À partir de maintenant, nous indiquons la clé en **gras**.

Chaque entité du schéma E/A devient une table de même nom dans la base de données, avec les mêmes attributs que l'entité. Étant donné le schéma E/A *Films*, on obtient les tables suivantes :

- Film (**id**, titre, année, genre, résumé)
- Artiste (**id**, nom, prénom, année_naissance)
- Internaute (**email**, nom, prénom, mot_de_passe, année_naissance)
- Pays (**code**, nom, langue)

On a perdu pour l'instant tout lien entre les relations.

4.2.2 Associations de un à plusieurs

On désigne par « associations de un à plusieurs » (que l'on abrège « associations 1 à n ») celles qui ont une cardinalité multiple d'un seul côté. Pour une association $A - \bullet B$, le passage à une représentation relationnelle suit les règles suivantes :

1. on crée les tables *A* et *B* correspondant aux entités ;
2. la clé de *A* devient *aussi* un attribut de *B*.

L'idée est qu'une ligne de *B* doit référencer une (et une seule) ligne de *A*. Cette référence peut se faire de manière unique et suffisante à l'aide de l'identifiant. On « exporte » donc la clé de *A* dans *B*, où elle devient une *clé étrangère*. Voici le schéma obtenu pour la base *Films* après application de cette règle. Les clés étrangères sont en *italiques*.

- Film (**id**, titre, année, genre, résumé, *id_réalisateur*, *code_pays*)
- Artiste (**id**, nom, prénom, année_naissance)
- Internaute (**email**, nom, prénom, mot_de_passe, année_naissance)
- Pays (**code**, nom, langue)

Il n'y a pas d'obligation à donner le même nom à la clé étrangère et la clé principale (que nous appellerons *clé primaire* à partir de maintenant). L'attribut *id_realisateur* correspond à l'attribut **id** d'*Artiste*, mais son nom est plus représentatif de son rôle exact : donner, pour chaque ligne de la table *Film*, l'identifiant de l'artiste qui a mis en scène le film.

Les tables ci-dessous montrent un exemple de la représentation des associations entre *Film* et *Artiste* d'une part, *Film* et *Pays* d'autre part (on a omis le résumé du film). Si on ne peut avoir qu'un artiste dont l'**id** est 2 dans la table *Artiste*, en revanche rien n'empêche cet artiste 2 de figurer plusieurs fois dans la colonne *id_realisateur* de la table *Film*. On a donc bien l'équivalent de l'association un à plusieurs élaborée dans le schéma E/A.

Tableau 4.5 — La table *Film*

id	titre	année	genre	*id_realisateur*	*code_pays*
1	Alien	1979	Science Fiction	51	US
2	Vertigo	1958	Suspense	52	US
3	Psychose	1960	Suspense	52	US
4	Kagemusha	1980	Drame	53	JP
5	Volte-face	1997	Action	54	US
6	Van Gogh	1991	Drame	58	FR
7	Titanic	1997	Drame	56	US
8	Sacrifice	1986	Drame	57	FR

Tableau 4.6 — La table *Artiste*

id	nom	prénom	année_naissance
51	Scott	Ridley	1943
52	Hitchcock	Alfred	1899
53	Kurosawa	Akira	1910
54	Woo	John	1946
56	Cameron	James	1954
57	Tarkovski	Andrei	1932
58	Pialat	Maurice	1925

Tableau 4.7 — La table *Pays*

code	nom	langue
US	Etats Unis	anglais
FR	France	français
JP	Japon	japonais

4.2.3 Associations de plusieurs à plusieurs

On désigne par « associations de plusieurs à plusieurs » (que l'on abrège en « associations n-m ») celles qui ont des cardinalités multiples des deux côtés. La transformation de ces associations est plus complexe que celle des associations un à plusieurs, ce qui explique que le choix fait au moment de la conception soit important. Prenons l'exemple de l'association *Joue* entre un artiste et un film. On ne peut pas associer l'identifiant *d'un* film à l'artiste, puisqu'il peut jouer dans plusieurs, et réciproquement on ne peut pas associer l'identifiant *d'un* acteur à un film.

En présence d'une association *A*•-•*B*, on doit donc créer une table spécifiquement destinée à représenter l'association elle-même, selon les règles suivantes :

1. on crée les tables *A* et *B* pour chaque entité ;
2. on crée une table *AB* pour l'association ;
3. la clé de cette table est la paire constituée des clés des tables *A* et *B* ;
4. les attributs de l'association deviennent des attributs de *AB*.

Pour identifier une association, on utilise donc la combinaison des clés des deux entités. Si on reprend la représentation sous forme de graphe, il s'agit en fait d'identifier le lien qui va d'une entité à une autre. Ce lien est uniquement déterminé par ses points de départ et d'arrivée, et donc par les deux clés correspondantes.

Voici le schéma obtenu après application de toutes les règles qui précèdent. On obtient deux nouvelles tables, *Rôle* et *Notation*, correspondant aux deux associations n-m du schéma de la figure 4.1.

- Film (**id**, titre, année, genre, résumé, *id_réalisateur*, *code_pays*)
- Artiste (**id**, nom, prénom, année_naissance)
- Internaute (**email**, nom, prénom, mot_de_passe, année_naissance)
- Pays (**code**, nom, langue)
- Rôle (***titre, id_acteur***, nom_rôle)
- Notation (***titre, email***, note)

Il peut arriver que la règle d'identification d'une association par les clés des deux entités soit trop contraignante quand on souhaite que deux entités puissent être liées plus d'une fois dans une association. Si, par exemple, on voulait autoriser un internaute à noter un film plus d'une fois, en distinguant les différentes notations par leur date, il faudrait, après avoir ajouté l'attribut `date`, identifier la table *Notation* par le triplet `(email, id_film, date)`. On obtiendrait le schéma suivant.

- Notation (*email*, ***id_film,*** **date**, note)

Il ne s'agit que d'une généralisation de la règle pour les associations n-m. Dans tous les cas, la clé est un sur-ensemble des clés des entités intervenantes.

Les tables suivantes montrent un exemple de représentation de *Rôle*. On peut constater le mécanisme de référence unique obtenu grâce aux clés des relations. Chaque rôle correspond à un unique acteur et à un unique film. De plus, on ne peut pas trouver deux fois la même paire `(titre,id_acteur)` dans cette table, ce qui n'aurait pas de sens. En revanche, un même acteur peut figurer plusieurs fois (mais pas associé au même film). Chaque film peut lui-aussi figurer plusieurs fois (mais pas associé au même acteur).

Tableau 4.8 — La table *Film*

id	titre	année	genre	*id_realisateur*	*code_pays*
9	Impitoyable	1992	Western	100	USA
10	Ennemi d'état	1998	Action	102	USA

Tableau 4.9 — La table *Artiste*

id	nom	prénom	année_naissance
100	Eastwood	Clint	1930
101	Hackman	Gene	1930
102	Scott	Tony	1930
103	Smith	Will	1968

Tableau 4.10 — La table *Rôle*

id_film	***id_acteur***	rôle
9	100	William Munny
9	101	Little Bill
10	101	Bril
10	103	Robert Dean

On peut remarquer finalement que chaque partie de la clé de la table *Rôle* est elle-même une clé étrangère qui fait référence à une ligne dans une autre table :

1. l'attribut `id_film` fait référence à une ligne de la table *Film* (un film) ;
2. l'attribut `id_acteur` fait référence à une ligne de la table *Artiste* (un acteur).

Le même principe de référencement et d'identification des tables s'applique à la table *Notation* dont un exemple est donné ci-dessous. Il faut bien noter que, par choix de conception, on a interdit qu'un internaute puisse noter plusieurs fois le même film, de même qu'un acteur ne peut pas jouer plusieurs fois dans un même film. Ces contraintes ne constituent pas des limitations, mais des décisions prises au moment de la conception sur ce qui est autorisé, et sur ce qui ne l'est pas. Vous devez, pour vos propres bases de données, faire vos propres choix en connaissance de cause.

Tableau 4.11 — La table *Film*

id	titre	année	genre	*id_realisateur*	*code_pays*
1	Alien	1979	Science Fiction	51	US
2	Vertigo	1958	Suspense	52	US
3	Psychose	1960	Suspense	52	US
4	Kagemusha	1980	Drame	53	JP
5	Volte-face	1997	Action	54	US
6	Van Gogh	1991	Drame	58	FR
7	Titanic	1997	Drame	56	US
8	Sacrifice	1986	Drame	57	FR

Tableau 4.12 — La table *Internaute*

email	nom	prénom	année_naissance
doe@void.com	Doe	John	1945
fogg@verne.fr	Fogg	Phileas	1965

Tableau 4.13 — La table *Notation*

id_film	***email***	note
1	fogg@verne.fr	4
5	fogg@verne.fr	3
1	doe@void.com	5
8	doe@void.com	2
7	fogg@verne.fr	5

Le processus de conception détaillé ci-dessus permet de décomposer toutes les informations d'une base de données en plusieurs tables, dont chacune stocke un des ensembles d'entités gérés par l'application. Les liens sont définis par un mécanisme de référencement basé sur les clés primaires et les clés étrangères. Il est important de bien comprendre ce mécanisme pour maîtriser la construction de bases de données qui ne nécessiteront par de réorganisation – nécessairement douloureuse – par la suite.

4.3 CRÉATION DE LA BASE *FILMS* AVEC MySQL

Il reste maintenant à créer cette base avec MySQL. La création d'un schéma comprend essentiellement deux parties : d'une part la description des *tables* et de leur contenu, d'autre part les *contraintes* qui portent sur les données de la base. La spécification des contraintes est souvent placée au second plan malgré sa grande importance. Elle permet d'assurer, au niveau de la base des contrôles sur l'intégrité des donnés qui s'imposent à toutes les applications accédant à cette base.

Le langage utilisé est la partie de SQL qui concerne la définition des données – le *Langage de Définition de Données* ou LDD. Il existe plusieurs versions de SQL. Le plus ancien standard date de 1989. Il a été révisé de manière importante en 1992. La norme résultant de cette révision, à laquelle se conforme MySQL, est SQL 92, SQL2 ou SQL ANSI. MySQL propose des extensions à la norme, mais nous allons nous fixer pour but de développer un site compatible avec tous les SGBD relationnels, ce qui amène à éviter ces extensions. Une discussion consacrée à la portabilité sur différents SGBD est proposée page 233.

4.3.1 Tables

La commande principale est `CREATE TABLE` que nous avons déjà rencontrée. Voici la commande de création de la table *Internaute*.

```
CREATE TABLE Internaute (email VARCHAR (40) NOT NULL,
                         nom VARCHAR (30) NOT NULL ,
                         prenom VARCHAR (30) NOT NULL,
                         mot_de_passe VARCHAR (32) NOT NULL,
                         annee_naissance INTEGER);
```

La syntaxe se comprend aisément. La seule difficulté est de spécifier le type de chaque attribut. MySQL propose un ensemble très riche de types, proche de la norme SQL ANSI. Nous nous limiterons à un sous-ensemble, suffisant pour la grande majorité des applications, présenté dans le tableau 4.14. Entre autres bonnes raisons de ne pas utiliser *tous* les types de MySQL, cela permet de rester compatible avec les autres SGBD. À l'exception de `TEXT`, les types mentionnés dans le tableau 4.14 sont connus de tous les SGBD relationnels.

Tableau 4.14 — Les principaux types de données SQL

Type	Description
`CHAR(`*n*`)`	Une chaîne de caractères de longueur fixe égale à *n*.
`INTEGER`	Un entier.
`VARCHAR(`*n*`)`	Une chaîne de caractères de longueur variable d'au plus *n*.
`DECIMAL(`*m*`,`*n*`)`	Un numérique sur *m* chiffres avec *n* décimales.
`DATE`	Une date, avec le jour, le mois et l'année.
`TIME`	Un horaire, avec heure, minutes et secondes.
`TEXT`	Un texte de longueur quelconque.

Le `NOT NULL` dans la création de table *Internaute* indique que l'attribut correspondant doit *toujours* avoir une valeur. Une autre manière de forcer un attribut à toujours prendre une valeur est de spécifier une *valeur par défaut* avec l'option `DEFAULT`.

```
CREATE TABLE Notation (id_film INTEGER NOT NULL,
                       email  VARCHAR (40) NOT NULL,
                       note   INTEGER DEFAULT 0);
```

4.3.2 Contraintes

La création d'une table telle qu'on l'a vue précédemment est assez sommaire car elle n'indique que le contenu de la table sans spécifier les contraintes que doit respecter ce contenu. Or il y a *toujours* des contraintes et il est indispensable de les inclure dans le schéma pour assurer, dans la mesure du possible, l'intégrité de la base.

Voici les règles – ou *contraintes d'intégrité* – que l'on peut demander au système de garantir :

1. un attribut doit toujours avoir une valeur ;
2. un attribut (ou un ensemble d'attributs) constitue(nt) la clé de la table ;
3. un attribut dans une table est lié à la clé primaire d'une autre table (*intégrité référentielle)* ;
4. la valeur d'un attribut doit être unique au sein de la table ;
5. un attribut ne peut prendre ses valeurs que parmi un ensemble prédéfini.

Les contraintes sur les clés doivent être systématiquement spécifiées.

Contrainte `NOT NULL`

Il peut arriver que la valeur d'un attribut soit inconnue : on dit dans ce cas qu'elle est « à `NULL` ». `NULL` n'est pas une valeur mais une absence de valeur ce qui est très différent d'une valeur « blanc » ou « 0 ». Conséquences :

1. on ne peut pas faire d'opération incluant un `NULL` ;
2. on ne peut pas faire de comparaison avec un `NULL`.

L'option `NOT NULL` oblige à toujours indiquer une valeur. On peut ainsi demander à MySQL de garantir que tout internaute a un mot de passe.

```
mot_de_passe VARCHAR(60) NOT NULL
```

MySQL rejettera alors toute tentative d'insérer une ligne dans *Internaute* sans donner de mot de passe. Si les valeurs à `NULL` sont autorisées, il faudra en tenir compte quand on interroge la base. Cela peut compliquer les choses, voire donner des résultats surprenants : forcez vos attributs important à avoir une valeur.

Clés d'une table

Il peut y avoir plusieurs clés dans une table, mais l'une d'entre elles doit être choisie comme *clé primaire*. Ce choix est important : la clé primaire est la clé utilisée pour référencer une ligne et une seule à partir d'autres tables. Il est donc assez délicat de la remettre en cause après coup. En revanche, les clés secondaires peuvent être créées ou supprimées beaucoup plus facilement. La clé primaire est spécifiée avec l'option `PRIMARY KEY`.

```
CREATE TABLE Internaute (email VARCHAR(40) NOT NULL,
                         nom VARCHAR(30) NOT NULL ,
```

```
    prenom VARCHAR (30) NOT NULL,
    mot_de_passe VARCHAR (32) NOT NULL,
    annee_naissance INTEGER,
    PRIMARY KEY (email));
```

Il devrait toujours y avoir une `PRIMARY KEY` dans une table pour ne pas risquer d'insérer involontairement deux lignes strictement identiques. Une clé peut être constituée de plusieurs attributs :

```
CREATE TABLE Notation (id_film INTEGER NOT NULL,
                       email  VARCHAR (40) NOT NULL,
                       note   INTEGER NOT NULL,
                       PRIMARY KEY (id_film , email));
```

Tous les attributs figurant dans une clé doivent être déclarés `NOT NULL`. Cela n'a pas vraiment de sens en effet d'identifier des lignes par des valeurs absentes. Certains SGBD acceptent malgré tout d'indexer des valeurs nulles, mais MySQL le refuse.

On peut également spécifier que la valeur d'un attribut est unique pour l'ensemble de la colonne. Cela permet d'indiquer des *clés secondaires*. On peut ainsi indiquer que deux artistes ne peuvent avoir les mêmes nom et prénom avec l'option `UNIQUE`.

```
CREATE TABLE Artiste  (id INTEGER NOT NULL,
                       nom VARCHAR (30) NOT NULL,
                       prenom VARCHAR (30) NOT NULL,
                       annee_naissance INTEGER,
                       PRIMARY KEY (id),
                       UNIQUE (nom, prenom));
```

Il est facile de supprimer cette contrainte de clé secondaire par la suite. Ce serait beaucoup plus difficile si on avait utilisé la paire `(nom, prenom)` comme clé primaire, puisqu'elle serait alors utilisée pour référencer un artiste dans d'autres tables.

La clé de la table *Artiste* est un numéro qui doit être incrémenté à chaque insertion. On pourrait utiliser l'option `AUTO_INCREMENT`, mais elle est spécifique à MySQL, ce qui empêcherait l'utilisation de notre application avec d'autres SGBD. Le site utilise un générateur d'identifiant décrit dans la section consacrée à la portabilité, page 233. Si vous ne vous souciez pas de compatibilité, l'utilisation de `AUTO_INCREMENT`, décrite page 72, est tout à fait appropriée.

Clés étrangères

La norme SQL ANSI permet d'indiquer les clés étrangères dans une table, autrement dit, quels sont les attributs qui font référence à une ligne dans une autre table. On peut spécifier les clés étrangères avec l'option `FOREIGN KEY`.

```
CREATE TABLE Film  (id INTEGER NOT NULL,
                    titre      VARCHAR (50) NOT NULL,
                    annee      INTEGER NOT NULL,
                    id_realisateur  INTEGER,
                    genre VARCHAR(30) NOT NULL,
```

```
resume          TEXT, /* LONG pour ORACLE */
code_pays       VARCHAR (4),
PRIMARY KEY (id),
FOREIGN KEY (id_realisateur) REFERENCES
    Artiste,
FOREIGN KEY (code_pays) REFERENCES Pays);
```

La commande

```
FOREIGN KEY (id_realisateur) REFERENCES Artiste
```

indique qu'`id_realisateur` référence la clé primaire de la table *Artiste*. En principe MySQL devrait vérifier, pour toute modification pouvant affecter le lien entre les deux tables, que la valeur de `id_realisateur` correspond bien à une ligne d'`Artiste`. Ces modifications sont :

1. l'insertion dans *Film* avec une valeur inconnue pour `id_realisateur` ;
2. la destruction d'un artiste ;
3. la modification d'`id` dans *Artiste* ou d'`id_realisateur` dans *Film*.

En d'autres termes le lien entre *Film* et *Artiste* est toujours valide. Cette contrainte est importante pour garantir qu'il n'y a pas de fausse référence dans la base, par exemple qu'un film ne fait pas référence à un artiste qui n'existe pas. Il est beaucoup plus confortable d'écrire une application par la suite quand on sait que les informations sont bien là où elles doivent être.

> **REMARQUE –** MySQL accepte *toujours* la clause `FOREIGN KEY`, mais n'applique les contraintes définies par cette clause que quand la table est créée avec le type `InnoDB`. InnoDB est un module de stockage et de gestion de transaction qui peut être utilisé optionnellement. Par défaut, MySQL crée des tables dont le type, MyISAM, est celui de son propre moteur de stockage, lequel ne reconnaît ni clés étrangères, ni transactions.

Énumération des valeurs possibles

La norme SQL ANSI comprend une option `CHECK (condition)` pour exprimer des contraintes portant soit sur un attribut, soit sur une ligne. La condition elle-même peut être toute expression suivant la clause `WHERE` dans une requête SQL. Les contraintes les plus courantes sont celles consistant à restreindre un attribut à un ensemble de valeurs, mais on peut trouver des contraintes arbitrairement complexes, faisant référence à d'autres relations.

Voici un exemple simple qui restreindrait, en SQL ANSI, les valeurs possibles des attributs `annee` et `genre` dans la table *Film*.

```
CREATE TABLE Film   (id INTEGER NOT NULL,
                     titre      VARCHAR (50) NOT NULL,
                     annee      INTEGER NOT NULL
                       CHECK (annee BETWEEN 1890 AND 2100) NOT
                           NULL,
                     id_realisateur   INTEGER,
```

```
    genre VARCHAR(30) NOT NULL
      CHECK (genre IN ('Histoire','Western','
          Drame')),
    resume        TEXT, /* LONG pour ORACLE */
    code_pays     VARCHAR (4),
    PRIMARY KEY (id),
    FOREIGN KEY (id_realisateur) REFERENCES
         Artiste,
    FOREIGN KEY (code_pays) REFERENCES Pays);
```

Comme pour les clés étrangères, MySQL accepte la clause `CHECK` mais ne traite pas la contrainte qu'elle définit. Il n'est pas non plus possible d'obtenir la contrainte définissant un intervalle pour les années.

Une autre manière de définir, dans la base, l'ensemble des valeurs autorisées pour un attribut –en d'autres termes, une codification imposée– consiste à placer ces valeurs dans une table et la lier à l'attribut par une contrainte de clé étrangère. C'est ce que nous avons fait par exemple pour la table *Pays*.

```
CREATE TABLE Pays (code     VARCHAR(4) NOT NULL,
                   nom  VARCHAR (30) DEFAULT 'Inconnu' NOT NULL,
                   langue VARCHAR (30) NOT NULL,
                   PRIMARY KEY (code));

INSERT INTO Pays (code, nom, langue)
          VALUES ('FR', 'France', 'Français');
INSERT INTO Pays (code, nom, langue)
          VALUES ('USA', 'Etats Unis', 'Anglais');
INSERT INTO Pays (code, nom, langue)
          VALUES ('IT', 'Italie', 'Italien');
. . .
```

Comme MySQL n'associe pas de vérification automatique à la commande `FOREIGN KEY` (du moins avec le type de tables par défaut), il faut faire cette vérification dans l'application, et notamment, comme nous le verrons, au niveau de l'interface qui permet de saisir les données.

Création de la base

Le fichier *Films.sql* donne le script complet de création de la base *Films*. À l'exception du type `TEXT` pour le résumé, les commandes sont conformes au SQL ANSI.

Exemple 4.1 *webscope/installation/Films.sql* : *Script de création de la base Films.*

```
/*   Commandes de création de la base Films.
      SQL ANSI SAUF le type TEXT (remplacer par LONG pour ORACLE) */

CREATE TABLE Internaute (email VARCHAR (40) NOT NULL,
                         nom VARCHAR (30) NOT NULL ,
                         prenom VARCHAR (30) NOT NULL,
```

```
                        mot_de_passe VARCHAR (32) NOT NULL,
                        annee_naissance INTEGER,
                        PRIMARY KEY (email));

CREATE TABLE Pays (code     VARCHAR(4) NOT NULL,
                   nom  VARCHAR (30) DEFAULT 'Inconnu' NOT NULL,
                   langue VARCHAR (30) NOT NULL,
                   PRIMARY KEY (code));

CREATE TABLE Artiste  (id INTEGER NOT NULL,
                       nom VARCHAR (30) NOT NULL,
                       prenom VARCHAR (30) NOT NULL,
                       annee_naissance INTEGER,
                       PRIMARY KEY (id),
                       UNIQUE (nom, prenom));

CREATE TABLE Film  (id INTEGER NOT NULL,
                    titre     VARCHAR (50) NOT NULL,
                    annee     INTEGER NOT NULL,
                    id_realisateur  INTEGER,
                    genre VARCHAR(30) NOT NULL,
                    resume       TEXT, /* LONG pour ORACLE */
                    code_pays     VARCHAR (4),
                    PRIMARY KEY (id),
                    FOREIGN KEY (id_realisateur) REFERENCES Artiste,
                    FOREIGN KEY (code_pays) REFERENCES Pays);

CREATE TABLE Notation (id_film INTEGER NOT NULL,
                       email  VARCHAR (40) NOT NULL,
                       note  INTEGER NOT NULL,
                       PRIMARY KEY (id_film, email),
                       FOREIGN KEY (id_film) REFERENCES Film,
                       FOREIGN KEY (email) REFERENCES Internaute);

CREATE TABLE Role (id_film INTEGER NOT NULL,
                   id_acteur INTEGER NOT NULL,
                   nom_role  VARCHAR(60),
                   PRIMARY KEY (id_film, id_acteur),
                   FOREIGN KEY (id_film) REFERENCES Film,
                   FOREIGN KEY (id_acteur) REFERENCES Artiste);
```

Ces tables sont créées, à l'aide du client *mysql*, avec la commande :

```
% mysql < Films.sql
```

en supposant, comme nous l'avons fait précédemment, que la base a été créée au préalable avec la commande `CREATE DATABASE Films`, et que l'utilisateur a son compte d'accès défini dans un fichier de configuration *.my.cnf*. On peut alors rappeler les options de création avec la commande `DESCRIBE`.

```
mysql> DESC Artiste;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| id              | int(11)     |      | PRI | 0       |       |
| nom             | varchar(30) |      | MUL |         |       |
| prenom          | varchar(30) |      |     |         |       |
| annee_naissance | int(11)     | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
```

4.3.3 Modification du schéma

La création d'un schéma n'est qu'une première étape dans la vie d'une base de données. On est toujours amené par la suite à créer de nouvelles tables, à ajouter des attributs ou à en modifier la définition. La forme générale de la commande permettant de modifier une table est :

```
ALTER TABLE nomTable ACTION description
```

où *ACTION* peut être principalement ADD, MODIFY, DROP ou RENAME, et *description* est la commande de modification associée à *ACTION*. La modification d'une table peut poser des problèmes si elle est incompatible avec le contenu existant. Par exemple, passer un attribut à NOT NULL implique que cet attribut a déjà des valeurs pour toutes les lignes de la table.

La commande DROP TABLE *nomTable* supprime une table. Elle est évidemment très dangereuse une fois la base créée, avec des données. *Il n'est plus possible de récupérer une table détruite avec* DROP TABLE.

Modification des attributs

Voici quelques exemples d'ajout et de modification d'attributs. La syntaxe complète de la commande ALTER TABLE est donnée dans l'annexe B.

On peut ajouter un attribut `region` à la table *Internaute* avec la commande :

```
ALTER TABLE Internaute ADD region VARCHAR(10);
```

S'il existe déjà des données dans la table, la valeur sera à NULL ou à la valeur par défaut. La taille de `region` étant certainement insuffisante, on peut l'agrandir avec MODIFY, et la déclarer NOT NULL par la même occasion :

```
ALTER TABLE Internaute MODIFY region VARCHAR(30) NOT NULL;
```

Il est également possible de diminuer la taille d'une colonne, avec le risque d'une perte d'information pour les données existantes. On peut même changer son type, pour passer par exemple de VARCHAR à INTEGER, avec un résultat non défini.

L'option ALTER TABLE permet d'ajouter une valeur par défaut.

```
ALTER TABLE Internaute ALTER region SET DEFAULT 'PACA';
```

Enfin on peut détruire un attribut avec DROP.

```
ALTER TABLE Internaute DROP region;
```

Voici une session de l'utilitaire *mysql* illustrant les commandes de mise à jour du schéma. phpMyAdmin propose de son côté des formulaires HTML très pratiques pour effectuer les mêmes modifications.

```
mysql> ALTER TABLE Internaute ADD region VARCHAR(10);
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC Internaute;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| email           | varchar(40) |      | PRI |         |       |
| nom             | varchar(30) |      |     |         |       |
| prenom          | varchar(30) |      |     |         |       |
| mot_de_passe    | varchar(32) |      |     |         |       |
| annee_naissance | int(11)     | YES  |     | NULL    |       |
| region          | varchar(10) | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+

mysql>  ALTER TABLE Internaute MODIFY region VARCHAR(30) NOT NULL;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC Internaute;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| email           | varchar(40) |      | PRI |         |       |
| nom             | varchar(30) |      |     |         |       |
| prenom          | varchar(30) |      |     |         |       |
| mot_de_passe    | varchar(32) |      |     |         |       |
| annee_naissance | int(11)     | YES  |     | NULL    |       |
| region          | varchar(30) | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+

mysql> ALTER TABLE Internaute ALTER region SET DEFAULT 'PACA';
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC Internaute;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| email           | varchar(40) |      | PRI |         |       |
| nom             | varchar(30) |      |     |         |       |
| prenom          | varchar(30) |      |     |         |       |
| mot_de_passe    | varchar(32) |      |     |         |       |
| annee_naissance | int(11)     | YES  |     | NULL    |       |
| region          | varchar(30) | YES  |     | PACA    |       |
+-----------------+-------------+------+-----+---------+-------+
```

```
mysql> ALTER TABLE Internaute DROP region;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Création d'index

Pour compléter le schéma d'une table, on peut définir des *index*. Un index offre un chemin d'accès aux lignes d'une table considérablement plus rapide que le balayage de cette table – du moins quand le nombre de lignes est très élevé. MySQL crée systématiquement un index sur la clé primaire de chaque table. Il y a deux raisons à cela ;

1. l'index permet de vérifier rapidement, au moment d'une insertion, que la clé n'existe pas déjà ;
2. beaucoup de requêtes SQL, notamment celles qui impliquent plusieurs tables (*jointures*), se basent sur les clés des tables pour reconstruire les liens. L'index peut alors être utilisé pour améliorer les temps de réponse.

Un index est également créé automatiquement pour chaque clause `UNIQUE` utilisée dans la création de la table. On peut de plus créer d'autres index, sur un ou plusieurs attributs, si l'application utilise des critères de recherche autres que les clés primaire ou secondaires.

La commande MySQL pour créer un index est la suivante :

```
CREATE [UNIQUE] INDEX nomIndex ON nomTable (attribut1 [, ...])
```

La clause `UNIQUE` indique qu'on ne peut pas trouver deux fois la même clé. La commande ci-dessous crée un index de nom *idxNom* sur les attributs `nom` et `prenom` de la table *Artiste*. Cet index a donc une fonction équivalente à la clause `UNIQUE` déjà utilisée dans la création de la table.

```
CREATE UNIQUE INDEX idxNom ON Artiste (nom, prenom);
```

On peut créer un index, cette fois non unique, sur l'attribut `genre` de la table *Film*.

```
CREATE INDEX idxGenre ON Film (genre);
```

Cet index permettra d'exécuter très rapidement des requêtes SQL ayant comme critère de recherche le genre d'un film.

```
SELECT *
FROM Film
WHERE genre = 'Western'
```

Cela dit il ne faut pas créer des index à tort et à travers, car ils ont un impact négatif sur les commandes d'insertion et de destruction. À chaque fois, il faut en effet mettre à jour tous les index portant sur la table, ce qui représente un coût certain.

5

Organisation du développement

Ce chapitre est une introduction aux choix techniques à effectuer au moment de la mise en développement d'un site basé sur PHP et MySQL. Avant de s'embarquer tête baissée dans la réalisation de scripts PHP, il est en effet important de se poser un certain nombre de questions sur la pertinence des décisions (ou des absences de décision...) prises à ce stade initial de développement, et sur leurs conséquences à court, moyen et long terme. Il s'agit véritablement d'envisager un changement d'échelle pour passer de la production de quelques scripts de petite taille comme ceux étudiés dans les chapitres précédents, à un code constitué de milliers de lignes utilisé quotidiennement par de nombreuses personnes et soumis à des évolutions produites par une équipe de développeurs. Voici un échantillon de ces questions :

1. comment organiser le code pour suivre une démarche logique de développement et de maintenance, et déterminer sans ambiguïté à quel endroit on doit placer tel ou tel fragment de l'application ;
2. quels outils utiliser pour tout ce qui relève du « génie logiciel » : édition des fichiers, sauvegardes, versions, livraisons, tests, etc.
3. comment assurer la portabilité à long terme et le respect des normes ?
4. quels sont les impératifs de sécurité, quel est le degré de robustesse et de confidentialité attendu ?

L'importance de ces questions est à relativiser en fonction du développement visé. Si vous êtes seul à produire et maintenir un site web dynamique basé sur quelques tables, quelques formulaires et un nombre limité de pages, le respect de quelques règles générales et l'utilisation d'outils légers suffira. Pour des applications professionnelles impliquant des équipes de développeurs pour plusieurs centaines de jours-homme planifiés, le recours à une méthodologie extrêmement rigoureuse s'impose. Dans ce dernier cas, il est d"ailleurs indispensable de s'appuyer sur un

framework de développement qui fournit un cadre de travail contraint et normalisé. Je présente un de ces frameworks, le *Zend Framework*, dans le chapitre 9.

Dans le présent chapitre nous allons commencer par tour d'horizon des régles organisationnelles de base, accompagné d'une présentation rapide des outils qui facilitent leur application. On peut très bien envisager de tout développer en utilisant le bloc-notes des accessoires Windows, mais il paraît plus sérieux de recourir à des outils spécialisés. Parmi les logiciels libres, il faut citer au minimum un environnement intégré comme *Eclipse*, un navigateur permettant de valider le code HTML comme *Firefox* associé à *Web Developer*, et enfin un système de gestion de versions comme Subversion ou CVS. Le réalisation de suites de tests avec PhpUnit et la production de documentation avec PhpDoc sont également brièvement abordés. Le but n'est pas ici de couvrir complètement des outils de génie logiciel, mais de montrer leur rôle et leur intérêt dans le cadre d'un processus de développement rigoureux.

Les sections suivantes sont consacrées à la résolution d'autres problèmes « structurels », indépendants des problèmes « fonctionnels » liés à une application spécifique : gestion des erreurs et des exceptions, et portabilité multi-SGBD. Ce livre ne prétend pas être un traité complet d'ingénierie logicielle, mais je propose pour chaque problème une solution, avec un double objectif : être à la fois concret, en fournissant une méthode utilisable, et simple, pour permettre au lecteur de comprendre la démarche.

Le prochain chapitre, complémentaire, sera consacré à l'organisation du code proprement dite, avec une introduction à l'architecture Modèle-Vue-Contrôleur (MVC), maintenant très souvent adoptée pour la réalisation d'applications web de taille significative.

5.1 CHOIX DES OUTILS

Voici pour commencer un bref aperçu de quelques valeurs sûres qui s'avèrent à l'usage extrêmement pratiques pour faciliter le développement et la maintenance d'un site.

5.1.1 Environnement de développement intégré Eclipse

L'écriture de code peut être assez rébarbative, et comprend de nombreux aspects répétitifs dont on peut penser qu'ils gagneraient à être automatisés. Les *Environnements de Développement Intégrés* (acronyme IDE en anglais) fournissent dans un cadre bien conçu tous les outils qui facilitent la tâche du développeur : contrôle syntaxique, navigation dans les fichiers, aide à la saisie, liste de tâches, etc. Le plus connu de ces IDE est certainement Eclipse (*http://www.eclipse.org*) initialement conçu et réalisé pour des applications Java, mais propose de très nombreuses extensions, dont une dédiée à PHP, le *PHP Development Tools* ou PDT.

La figure 5.1 montre Eclipse en action avec la perspective PDT sur le site WEBSCOPE. L'ensemble des fenêtres et leur disposition sont entièrement configurables. Voici une description qui vous donnera une idée de la puissance de cet outil.

- la partie gauche supérieure montre la hiérarchie des répertoires du projet WEBSCOPE ;
- la partie gauche inférieure est une aide à la programmation PHP, avec entre autres la possibilité de trouver rapidement une fonction et son mode d'appel ;
- la partie centrale est le fichier PHP en cours d'édition ; les catégories syntaxiques (variables, instructions, structures de contrôle) sont mises en valeur par des codes couleurs, et les erreurs de syntaxe sont détectées et signalées par l'éditeur ;
- la partie droite est un résumé de la structure du fichier PHP courant ; ici il s'agit d'une classe, avec des méthodes privées et publiques, des constantes, des propriétés, etc. ;
- enfin la partie basse contient des informations sur le projet et le fichier courant, comme les tâches à effectuer, des annotations sur le code, la liste des problèmes détectés, etc.

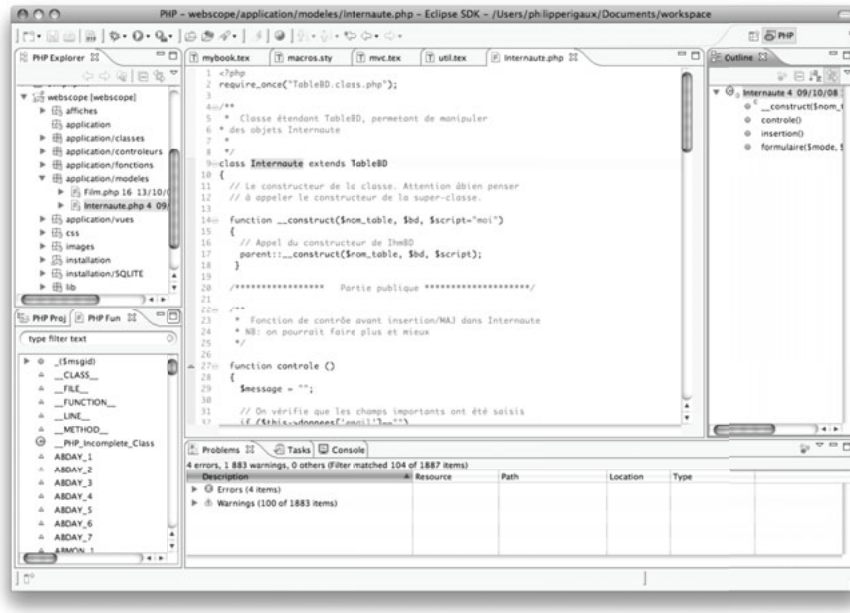


Figure 5.1 — Environnement de développement Eclipse pour PHP

L'apprentissage de ce type d'outil demande quelques heures d'investissement pour une utilisation basique ou quelques jours pour une utilisation avancée. Dans tous les cas, le gain en termes de confort d'utilisation et de temps est considérable. Je ne saurais donc trop vous conseiller d'effectuer cet effort dès que vous entamerez la réalisation de scripts PHP qui dépassent les simples exemples vus jusqu'à présent.

L'installation est simple (et gratuite). Vous devez commencer par installer Eclipse, téléchargeable sur le site *http://www.eclipse.org*. Pour l'extension PHP, toutes les instructions se trouvent sur la page *http://www.eclipse.org/pdt/*. Essentiellement il suffit,

dans Eclipse, d'accéder au choix *Software update* du menu *Help*, et de télécharger PDT à partir de *http://download.eclipse.org/tools/pdt/updates*. En cas de problème, vérifiez les dépendances et compatibilités de version en cherchant sur le Web : on trouve presque toujours quelqu'un qui a bien voulu indiquer la marche à suivre.

5.1.2 Développement en équipe : Subversion

Si vous développez seul, une seule version de vos fichiers sur votre machine suffit. Dès que l'on travaille à plusieurs sur la même application, le problème des mises à jour concurrentes se pose. Comment être sûr qu'on ne va pas se retrouver à travailler à deux sur le même fichier, avec risque de conflit ; comment récupérer facilement les évolutions effectuées par quelqu'un d'autre ; comment gérer des versions, surveiller les évolutions, comprendre ce qui a changé ? Des outils ont été créés pour faciliter la gestion des évolutions et le développement collectif sur une même application. Le plus répandu est sans doute CVS (*Concurrent Versioning System*), qui tend à être remplacé par Subversion, un autre système très proche dans ses principes mais un peu plus puissant.

La présentation des principes de gestion de version dépasse évidemment le cadre de ce livre[1], mais il est important d'être au moins averti de l'existence de ces outils, de leur apport à la résolution des problèmes soulevés par le développement coopératif, et enfin de leur facilité de mise en œuvre. Une fois la configuration effectuée, un ou deux clics suffisent pour livrer les évolutions effectuées, et au contraire récupérer les évolutions faites par d'autres.

Vous pouvez tout à fait sauter la description qui suit si cette problématique ne vous concerne pas, ou pas tout de suite. Mais si vous êtes intéressés par la découverte et l'expérimentation d'un développement en commun et d'utilisation de CVS, je vous propose tout simplement de participer à l'amélioration du site WEBSCOPE dont le code est disponible sur le serveur CVS de *SourceForge* à l'adresse suivante.

webscope.cvs.sourceforge.net

Voici comment procéder, en utilisant Eclipse qui fournit une interface de navigation et d'utilisation de CVS simple et puissante[2]. Il faut tout d'abord indiquer le serveur CVS. Pour cela, accédez au menu *Windows*, puis *Open perspective* et choisissez la perspective CVS. La fenêtre de gauche montre alors la liste des serveurs CVS répertoriés. Elle est initialement vide, mais vous allez ajouter un serveur avec le bouton CVS situé en haut de la fenêtre. La figure 5.2 montre le formulaire de configuration qui s'affiche alors.

Entrez les informations comme indiqué. Pour le compte de connexion, vous pouvez soit utiliser une connexion anonyme si vous n'avez pas créé de compte sur

1. Je vous recommande la lecture du livre (en anglais) librement disponible consacré à SubVersion, à l'adresse *http://svnbook.red-bean.com/*.
2. CVS est nativement intégré à Eclipse. Pour utiliser Subversion il faut installer Subclipse, ce qui se fait en quelques minutes.

Figure 5.2 — Configuration de la connexion au serveur CVS

SourceForge, soit utiliser votre compte SourceForge. Dans le premier cas vous pourrez juste récupérer le code, sans faire de modification. Il est bien entendu préférable de créer un compte sur SourceForge pour bénéficier pleinement des fonctionnalités collaboratives.

Une fois connecté au serveur CVS, vous pouvez explorer les versions et les fichiers du projet WEBSCOPE. La figure 5.3 montre la navigation et la consultation des

Figure 5.3 — Exploration du répertoire distant CVS

fichiers dans la branche `HEAD` qui contient la version en cours de développement du projet. Les versions successives sont dans d'autres branches.

Vous pouvez récupérer une version en utilisant le clic droit sur un répertoire (par exemple, *webscope* de la branche `HEAD`) et en choisissant l'option `checkout`. Eclipse va alors importer l'ensemble des fichiers du site dans un projet sur votre machine locale, et vous pouvez commencer des modifications sur les fichiers pour améliorer le code. Toutes les modifications agissent sur la version locale, indépendamment de tout ce qui peut se passer par ailleurs sur le serveur CVS de SourceForge. Quand vous estimez que vous avez apporté une contribution significative au code et que vous souhaitez l'intégrer au CVS, utilisez à nouveau le clic droit sur votre projet local, et choisissez l'option *Team*, puis *Commit* comme indiqué sur la figure 5.4.

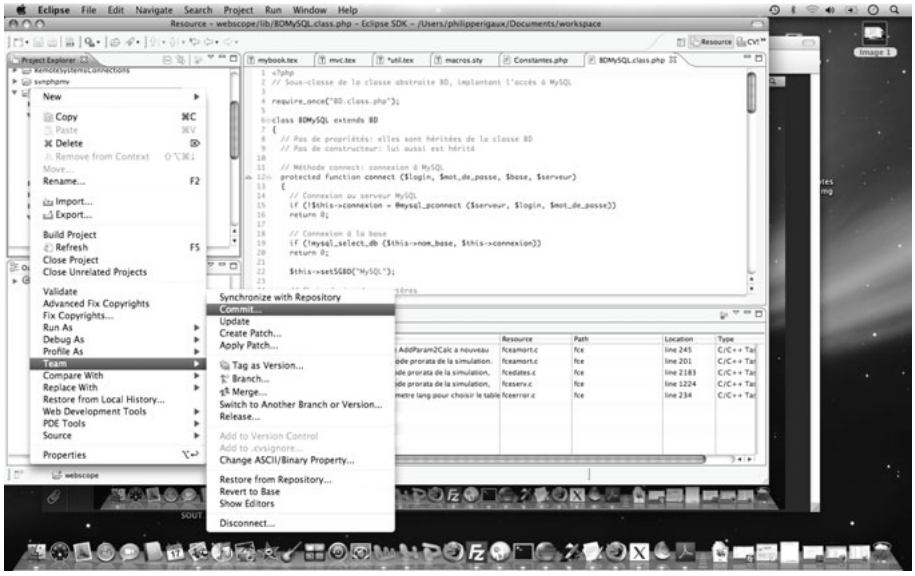


Figure 5.4 — Validation de modifications, et transfert sur le serveur CVS

Vous voici entré dans le processus de développement coopératif avec Eclipse et CVS. À chaque moment, vous pouvez au choix utiliser la commande *Commit* pour valider vos modifications et les transférer sur le CVS, ou au contraire la commande *Update* pour récupérer dans votre copie locale les modifications effectuées par les autres utilisateurs.

Je n'en dis pas plus à ce stade. Lisez un tutorial sur CVS pour comprendre le fonctionnement de base (qui tient en quelques commandes) et pratiquez avec le site CVS que je vous propose sur SourceForge. Le site web du livre vous informera des développements et évolutions de ce prolongement collectif au code décrit dans le reste de ce livre.

5.1.3 Production d'une documentation avec PhpDoc

La communauté des développeurs PHP a produit de nombreux outils pour constituer des environnements logiciels de qualité. Ces outils contribuent à faire de PHP un concurrent tout à fait présentable de langages anciens et éprouvés comme C++ ou Java. La possibilité de produire une documentation directement à partir du code fait partie de ces acquis. Dans le monde PHP, l'outil qui semble le plus utilisé est PhpDocumentor *http://www.phpdoc.org/* et c'est donc celui que je présente ensuite. Cela étant des logiciels plus généralistes comme *doxygen*, qui s'applique également au C, au C++, à Java et à beaucoup d'autres langages, produisent également un très beau travail.

Documenter du PHP pour PhpDoc

PhpDoc produit un site HTML statique contenant une description technique extraites des fichiers PHP d'une application web. La figure 5.5 montre un exemple d'une page PhpDoc produite automatiquement pour le site WEBSCOPE.

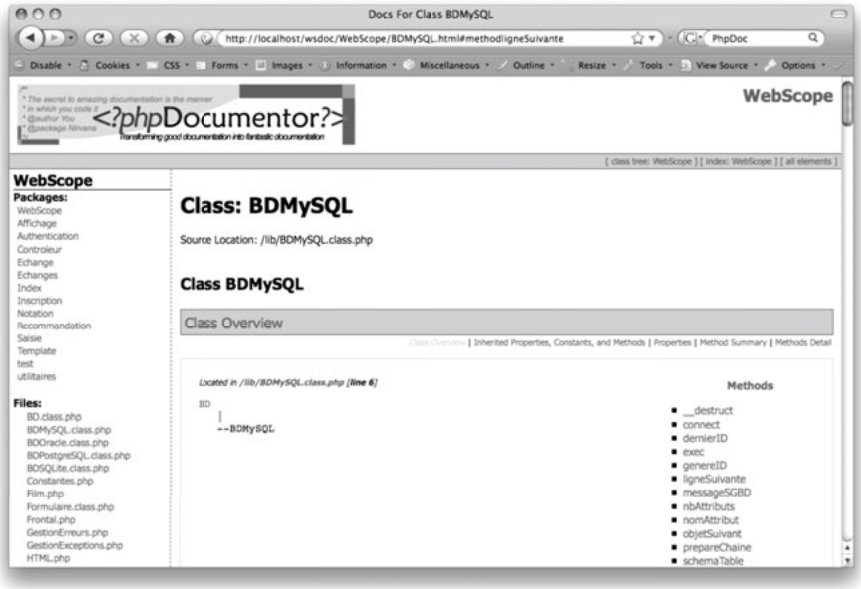


Figure 5.5 — Exemple de page produite par PhpDoc

La documentation est basée sur la notion de *DocBlock* qui sert à documenter des « éléments » du code. Les éléments sont les fonctions, les variables, les classes, et tous les composants logiciels d'une application PHP. Chaque DocBlock est simplement un commentaire de la forme `/** ...*/` (notez les deux étoiles initiales) constitué de trois parties aparaissant dans l'ordre suivant :

1. une description courte ;
2. une description longue ;

3. des balises choisies parmi un ensemble pré-défini et décrivant un des aspects de l'élément documenté (par exemple, la balise `@author` indique l'auteur de l'élément).

La stratégie utilisée pour la documentation varie selon le type d'élément documenté. Pour faire simple, limitons-nous ici au cas des classes PHP orientées-objet. On peut les documenter à deux niveaux : la classe et la méthode (on pourrait envisager trois niveaux si on mettait plusieurs classes dans une page). Voici quelques exemples de balises utiles dans ce contexte.

- `@category` est le nom de l'application ;
- `@package` est une notion correspondant à un regroupement de classes partageant un même objectif (par exemple toutes les classes interagissant avec la base de données) ;
- `@copyright` est le nom du titulaire de la propriété intellectuelle ;
- `@license` est la licence d'exploitation ;
- `@version` est le numéro de version.

Voici un exemple de DocBlock pour la classe `BD` de notre application.

```
/**
 *  Classe abstraite définissant une interface générique d'accès à une BD
 *
 *  Cette classe définit les méthodes génériques d'accès à une base de données
 *  quel que soit le serveur utilisé. Elle est abstraite et doit être spécialisée
 *  pour chaque système (MySQL, Oracle, etc.)
 *
 *  @category Pratique de MySQL et PHP
 *  @package BD
 *  @copyright Philippe Rigaux
 *  @licence GPL
 *  @version 1.0.0
 */
```

Au niveau des méthodes, on peut ajouter la description du type et du rôle de chaque paramètre, ainsi que le type de la valeur retournée. Les paramètres sont marqués par la balise `@param`, suivi du type et d'une phrase qui décrit le paramètre. La balise `@tag` suit a même convention. Voici un exemple, toujours tiré de la classe `BD`.

```
 /**
   *  Constructeur de la classe
   *
   *  Le constructeur appelle la méthode connect() de la classe
   *  et vérifie que la connexion a bien été établie. Sinon une
   *  exception est levée.
   *
   * @param string Login de connexion
   * @param string mot de passe
```

```
   * @param string nom de la base
   * @param string nom du serveur
   * @return null
   */

  function __construct ($login, $mot_de_passe, $base, $serveur)
  {
    .. ..
   }
```

La production de cette documentation technique est particulièrement utile pour les bibliothèques, classes et fonctions utilitaires fréquemment appelées et pour lesquelles une description des modes d'appel est indispensable.

Comment utiliser PhpDoc

PhpDoc s'installe très simplement comme une application PHP. Récupérez sur *http://www.phpdoc.org/* le fichier archive et décompressez-le dans le répertoire *htdocs*. Renommez le nouveau répértoire obtenu en *phpdoc*. Vous pouvez maintenant y accéder à *http://localhost/phpdoc*.

Si vous voulez documenter une application, par l'exemple l'application WEBSCOPE, le plus simple, pour éviter de saisir systématiquement les paramètres de production de la documentation, est de créer un fichier de configuration à placer dans *users/* dans le répertoire *phpdoc*. À titre d'illustration, voici un fichier de configuration minimal permettant d'analyser l'application web WEBSCOPE et de placer la documentation générée dans *wsdoc*.

```
;Titre général
title = Documentation WebScope

;; Quelle est l'application à documenter
directory = /Applications/MAMP/htdocs/webscope

;; Où écrire la documentation?
target = /Applications/MAMP/htdocs/wsdoc

;;Doit-on considérer les fichiers cachés?
hidden = false

;; Doit-on montrer les éléments privés? (@access private)
parseprivate = off

;; Quel est le package principal?
defaultpackagename = WebScope

;; Fichiers à ignorer
ignore = *.tpl

;; Style de la documentation
output=HTML:Smarty:HandS
```

Ce fichier de configuration apparaît alors dans la liste des choix quand on accède à la page de configuration de PhpDoc. Il ne reste plus ensuite qu'à l'afficher avec le navigateur web. PhpDoc peut également engendrer d'autres formats, et notamment le format DocBook qu'on peut ensuite transformer en PDF. Toutes les documentations techniques des composants PHP Open Source sont créées de cette manière (mais pas toujours avec PhpDoc, car, comme signalé ci-dessus, des logiciels comme *doxygen* font un travail au moins équivalent et valent la peine d'être étudiés).

5.1.4 Tests unitaires avec PhpUnit

Vous devez bien entendu tester vos développements et vous assurer de leur correction, en toutes circonstances. Le test est une tâche fastidieuse mais nécessaire pour une production de qualité. Le contrôle et la certification du logiciel constituent un sujet extrêmement vaste. Une première étape consiste à effectuer des *test unitaires* afin de contrôler les briques de base d'une application, si possible de façon automatique.

L'outil de test unitaire pour PHP s'appelle *PhpUnit* et constitue la déclinaison pour PHP de JUnit (pour Java) ou CppUnit (pour C++). Son site d'accueil est *http://www.phpunit.de*. Ce qui suit constitue une brève introduction à son utilisation.

Il faut commencer par installer PhpUnit. Le site donne deux procédures d'installation : la première avec *pear*, un gestionnaire de composants PHP, la seconde par téléchargement et configuration. Si *pear* n'est pas installé dans votre environnement, suivez simplement les instructions sur le site de PHPUnit pour une installation directe.

Dans les deux cas, on se retrouve avec un script PHP *phpunit* qui s'exécute en ligne de commande (pas d'interface web). Commençons par un exemple trivial. Nous avons créé une classe `Addition` avec une méthode `ajout()` dont le but est d'ajouter deux nombres. Le code n'est pas trop compliqué :

Exemple 5.1 *exemples/Addition.php* : *Une classe sans intérêt, mais à tester quand même*

```
<?php

class Addition {
  public function ajout ($a, $b)
  {
      return $a + $b;
  }
}

?>
```

Maintenant nous allons créer un second script PHP qui va tester le premier. Comme il s'agit d'un cas fictif, les deux scripts sont dans le répertoire de nos exemples, mais en général il faut bien entendu imaginer que l'application de test est séparée de celle qui est testée.

Exemple 5.2 *exemples/PremierTest.php* : *Une seconde classe, qui teste la première*

```
<?php

/** Test de la classe addition
 *
 */

require_once ('PHPUnit/Framework.php');
require_once ("Addition.php");

class PremierTest extends PHPUnit_Framework_Testcase {

  public function testAjout() {
    $addition = new Addition();
    $this->assertEquals (2, $addition->ajout(1, 1));
    $this->assertNotEquals(3, $addition->ajout(2, 2));
  }
}
?>
```

La simplicité de l'exemple a le mérite de le rendre assez clair. La classe de test instancie un objet de la class testée, exécute une méthode et effectue des contrôles sur le résultat obtenu. On vérifie ici que $1 + 1 = 2$ et que $2 + 2 \neq 3$. Il reste à lancer le script *phpunit* sur cette classe de test.

```
> phpunit PremierTest
PHPUnit 3.3.1 by Sebastian Bergmann.

.

Time: 0 seconds

OK (1 test, 2 assertions)
```

Tout s'est bien passé. Voici maintenant quelques explications. PHPUnit s'appuie sur des conventions de nommage consistant à donner aux classes de test un nom se terminant par `Test` et aux méthodes de test un nom commençant par `test`. La classe de test ne doit pas être située dans le même répertoire que l'application : le but est de lancer une application (de test) qui travaille sur une autre application (normale), cette dernière ne devant pas subir la moindre modification.

Une classe de test hérite de `PHPUnit_FrameworkTestCase`. Ce faisant elle dispose de tout un ensemble d'*assertions* et de mécanismes pour exécuter les tests. Le script *phpunit* reçoit le nom de la classe de test et exécute chacune des méthodes de test. À l'intérieur de chaque méthode de test, on place une liste d'assertions exprimant ce que le code testé doit faire et quels résultats il doit fournir. Dans notre exemple trivial, on vérifie les résultats de deux additions. Dans un exemple plus réaliste, il faut inclure toutes les assertions exprimant ce qui doit caractériser selon

nous le comportement de la méthode testée. À titre d'exemple, changez le + en - dans notre méthode d'addition, puis effectuez à nouveau le test. Voici ce que l'on obtient :

```
> phpunit PremierTest
PHPUnit 3.3.1 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) testAjout(PremierTest)
Failed asserting that <integer:0> matches expected value <integer:2>.
/Applications/MAMP/htdocs/exemples/PremierTest.php:14

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Un des tests sur la méthode `ajout()` a échoué (celui qui effectue le contrôle $2 = 1 + 1$), l'autre a réussi (celui qui vérifie que $3 \neq 2 + 2$). Il existe bien entendu de très nombreuses autres assertions que vous pouvez découvrir dans la documentation de PHPUnit.

Effectuer des tests implique d'instancier la classe à tester, puis d'appliquer des méthodes sur l'objet obtenu. Pour éviter l'aspect répétitif de ce mécanisme, PHPUnit fournit un générateur de « squelette » d'une classe de test. La commande, toujours sur notre exemple simple, est :

```
> phpunit --skeleton Addition
```

On obtient une classe `AdditionTest` que voici :

Exemple 5.3 *exemples/AdditionTest.php* : *La classe de test engendrée automatiquement par PHPUnit*

```
<?php
require_once 'PHPUnit/Framework.php';

require_once 'Addition.php';

/**
 * Test class for Addition.
 * Generated by PHPUnit on 2008-10-19 at 17:36:45.
 */
class AdditionTest extends PHPUnit_Framework_TestCase
{
    /**
     * @var     Addition
     * @access  protected
```

```
     */
    protected $object;

    /**
     * Sets up the fixture, for example, opens a network
         connection.
     * This method is called before a test is executed.
     *
     * @access protected
     */
    protected function setUp()
    {
        $this->object = new Addition;
    }

    /**
     * Tears down the fixture, for example, closes a network
         connection.
     * This method is called after a test is executed.
     *
     * @access protected
     */
    protected function tearDown()
    {
    }

    /**
     * @todo Implement testAjout().
     */
    public function testAjout() {
        // Remove the following lines when you implement this
            test.
        $this->markTestIncomplete(
          'This test has not been implemented yet.'
        );
    }
}
?>
```

Deux méthodes spéciales, `setUp()` et `tearDown()` ont été créées pour, respectivement, instancier un objet de la classe `Addition` et libérer cet environnement de test. C'est à nous de compléter ces deux méthodes pour initialiser l'environnement de test (par exemple on pourrait se connecter à la base de données avant d'effectuer des tests sur une application PHP/MySQL). Ensuite PHPUnit crée une méthode `test`*`nomMéth`*`()` pour chaque méthode *`nomMéth`* de la classe testée. Ici nous avons donc une méthode `testAjout()`. Toutes ces méthodes de test sont à implanter, comme le montre le `@todo` placé dans le DocBlock.

Quand ce travail est réalisé pour toutes les classes et fonctions d'une application, on peut regrouper les tests dans des *suites* grâce à la classe

`PHPUnit_FrameworkTestSuite`. Voici un exemple simple montrant comment intégrer notre classe de tests dans une suite.

Exemple 5.4 *exemples/MesTests.php* : *Création d'une suite de tests*

```
<?php
/**
 * Ensemble des tests de l'application
 */

require_once 'PHPUnit/Framework.php';
require_once 'PHPUnit/TextUI/TestRunner.php';

/** Inclusion des classes à tester
 *
 */
require_once 'AdditionTest.php';

class MesTests
{
  public static function main()
  {
    PHPUnit_TextUI_TestRunner::run(self::suite());
  }

  public static function suite()
  {
    $suite = new PHPUnit_Framework_TestSuite('Tous mes tests');
    $suite->addTestSuite("AdditionTest");
    return $suite;
  }
}

?>
```

On peut ensuite exécuter une suite de tests avec *phpunit*. Arrêtons là pour cette brève introduction dont le but est esentiellement de vous donner une idée du processus de constitution de tests automatiques pour valider une application. Une fois ces tests mis en place – ce qui peut évidemment prendre beaucoup de temps – on peut les ré-exécuter à chaque nouvelle version de l'application pour vérifier qu'il n'y a pas de régression.

5.1.5 En résumé

Ce qui précède a montré une partie des outils qui constituent un environnement de haut niveau pour la production et la maintenance d'applications web. On pourrait encore citer *Phing*, un descripteur de tâches comparable au *make* Unix, pour enchaîner automatiquement des étapes de construction (vérification syntaxique, tests,

documentation, etc.) d'une application livrable, *Xdebug* pour déverminer (« déboguer » . . .) ou profiler des applications, etc.

Encore une fois l'utilisation de ces outils est à apprécier en fonction du contexte. Eclipse est vraiment un *must* : cet IDE rend de tels services qu'il est vraiment difficile de s'en passer une fois qu'on y a goûté. Les tests et la documentation constituent quant à eux des efforts importants qui s'imposent principalement dans les processus de production de code de haute qualité, en vue par exemple d'une certification.

5.2 GESTION DES ERREURS

Même si l'on a mis en place une procédure de tests automatisée avec PHPUnit, il faut toujours envisager qu'une erreur survienne pendant le déroulement d'une application. La gestion des erreurs est un problème récurrent. Il faut se poser en permanence la question des points faibles du code et des conséquences possibles d'un fonctionnement incorrect ou de données non conformes à ce qui est attendu. Cette vigilance est motivée par trois préoccupations constantes :

1. avertir correctement l'utilisateur du problème et des solutions pour le résoudre ;
2. ne pas laisser l'application poursuivre son exécution dans un contexte corrompu ;
3. être prévenu rapidement et précisément de la cause de l'erreur afin de pouvoir la corriger.

Il faut également s'entendre sur le sens du mot « erreur ». Nous allons en distinguer trois types : erreurs d'utilisation, erreurs syntaxiques et erreurs internes.

Erreurs d'utilisation

Dans le contexte d'applications web, de nature fortement interactives, beaucoup « d'erreurs » résultent de données ou d'actions imprévues de la part de l'utilisateur. Ce dernier n'est pas en cause, puisqu'on peut très bien considérer que l'interface devrait interdire ces saisie ou actions. Il n'en reste pas moins que ces erreurs se caractérisent par la nécessité de fournir un retour indiquant pourquoi l'appel à telle ou telle fonctionnalité a été refusé ou a échoué.

Nous avons déjà étudié la question du contrôle des données en entrée d'un script (voir page 70) et la production de messages en retour. Toute erreur d'utilisation implique une communication avec l'utilisateur, laquelle prend dans la majorité des cas la forme d'un message à l'écran.

Erreurs internes

Les erreurs internes les plus communes sont dues à la manipulation de données anormales (comme une division par zéro) ou à la défaillance d'un des composants de l'application (le serveur de base de données par exemple). Ce qui caractérise

une erreur interne, c'est l'apparition d'une configuration dans laquelle l'application ne peut plus fonctionner correctement. Ces configurations ne sont pas toujours détectables durant la phase de test, car elles dépendent parfois d'événements qui apparaissent de manière imprévisible. Une bonne application devrait être capable de réagir correctement à ce type d'erreur.

Erreurs syntaxiques

Enfin, les erreurs syntaxiques sont dues à une faute de programmation, par exemple l'appel à une fonction avec de mauvais paramètres, ou toute instruction incorrecte empêchant l'interprétation du script. En principe, elles devraient être éliminées au moment des tests. Si ceux-ci ne sont pas menés systématiquement, certaines parties du code peuvent ne jamais être testées avant le passage en production.

L'approche PHP

La section qui suit présente les principales techniques de traitement d'erreur en PHP. Les *erreurs d'utilisation* ne sont pas spécifiquement considérées puisque nous avons déjà vu de nombreux exemples, et qu'il n'y a pas grand chose d'autre à faire que de tester systématiquement les entrées d'un script ou d'une fonction, et de produire un message si quelque chose d'anormal est détecté. L'utilisation des exceptions PHP n'est pas pratique dans ce cas, car un lancer d'exception déroute le flux d'exécution du script vers la prochaine instruction `catch`, ce qui n'est souvent pas souhaitable pour ce type d'erreur. Les *erreurs syntaxiques* doivent être éliminées le plus vite possible. La première sous-section ci-dessous montre comment mettre en œuvre dès la phase de développement un contrôle très strict des fautes de programmation.

Enfin les *erreurs internes* peuvent être interceptées et traitées, en PHP 5, par l'un ou l'autre des deux moyens suivants :

1. les *erreurs PHP* ;
2. les *exceptions*.

Pour chacun il est possible de définir des gestionnaires d'erreur spécialisés, que l'on pourra donc régler différemment sur un site de développement ou sur un site de production.

5.2.1 Erreurs syntaxiques

Les fautes de programmation sont en principe détectables au moment des tests, si ceux-ci sont menés de manière suffisamment exhaustive. PHP est un langage assez permissif, qui autorise une programmation assez relâchée. Cela permet un développement très rapide et assez confortable, mais en contrepartie cela peut dans certains cas rendre le comportement du script erroné.

En PHP les variables ne sont pas déclarées, sont typées en fonction du contexte, et peuvent même, si l'installation est configurée assez souplement, ne pas être

initialisées. Dans beaucoup de cas, l'interpréteur PHP essaie de corriger automatiquement les imprécisions ou erreurs de syntaxe légères dans un script. Voici un exemple d'un script contenant beaucoup de minimes incorrections syntaxiques. En supposant que PHP est configuré dans un mode où la non-déclaration des variables est tolérée, la correction s'effectue silencieusement, avec des résultats parfois insatisfaisants.

Exemple 5.5 *exemples/TestErreur.php* : *Un script avec des erreurs minimes de code.*

```
<?php
 // Script montrant l'usage du contrôle des erreurs

  header ("Content-type: text/plain");

  define(ma_constante, 5);

  $tableau = array ("1" => "Valeur 1",
                    "second_element" => "Valeur 2",
                    "ma_constante" => "Valeur 3");
  $texte = "Un texte à afficher";

  echo "Affichage de la variable \$texte : $texTe\n";
  echo "Premier élément = " . $tableau[1] . "\n";
  echo "Second élément = " . $tableau[second_element] . "\n";
  echo "Dernier élément = " . $tableau[ma_constante];
?>
```

Ce script se contente de produire du texte non HTML. Voici ce qui s'affiche dans la fenêtre du navigateur :

```
Affichage de la variable $texte :
Premier élément = Valeur 1
Second élément = Valeur 2
Dernier élément =
```

Ce n'est pas tout à fait ce qui était souhaité. Le contenu de la variable `$texte` et celui du dernier élément du tableau ne s'affichent pas (voyez-vous d'où vient le problème ?). Ce genre d'anomalie peut passer inaperçu, ou être très difficile à détecter.

Il est possible de régler le niveau des messages d'erreur produits par PHP avec la fonction `error_reporting()` qui prend en argument un ou plusieurs des niveaux de messages du tableau 5.1.

Ces niveaux sont des constantes prédéfinies qui peuvent être combinées par des opérateurs de bits (voir page 429). L'appel à la fonction `error_reporting()` avec l'argument `E_ERROR | E_WARNING` demande l'affichage des deux types d'erreur. La valeur par défaut [3] est généralement `E_ALL | ~E_NOTICE` ce qui signifie que toutes

3. Elle dépend de l'installation de PHP.

Tableau 5.1 — Niveau des messages d'erreur dans PHP

Valeur	Niveau d'erreur	Description
	E_ALL	Tous les avertissements et erreurs ci-dessous.
1	E_ERROR	Erreurs fatales (interruption du script).
2	E_WARNING	Erreurs légères (le script continue).
4	E_PARSE	Erreur de compilation/analyse.
8	E_NOTICE	Avertissements (une erreur légère qui peut être intentionnelle, comme la non-initialisation d'une variable).
16	E_CORE_ERROR	Erreurs fatales pendant le lancement de PHP.
32	E_CORE_WARNING	Avertissement pendant le lancement de PHP.
64	E_COMPILE_ERROR	Erreur fatale pendant la compilation.
128	E_COMPILE_WARNING	Avertissement pendant la compilation.
256	E_USER_ERROR	Erreur fatale engendrée par le programmeur.
512	E_USER_WARNING	Erreur légère engendrée par le programmeur.
1024	E_USER_NOTICE	Avertissement engendré par le programmeur.
1024	E_STRICT	Avertissement indiquant une syntaxe PHP 4 qui risque de ne plus être supportée à l'avenir.

les erreurs sont signalées, sauf les « avertissements ». Voici ce que l'on obtient avec le script précédent en plaçant au début un appel à `error_reporting()` avec la valeur `E_ALL` :

```
<b>Notice</b>:  Use of undefined constant ma_constante -
  assumed 'ma_constante' in <b>TestErreur.php</b> on line <b>8</b>

<b>Notice</b>:  Undefined variable:  texTe in
 <b>TestErreur.php</b> on line <b>15</b>

Affichage de la variable $texte :
Premier élément = Valeur 1

<b>Notice</b>:  Use of undefined constant second_element -
  assumed 'second_element' in <b>TestErreur.php</b> on line <b>17</b>

Second élément = Valeur 2

<b>Notice</b>:  Undefined offset:  5 in
 <b>TestErreur.php</b> on line <b>18</b>

Dernier élément =
```

Quatre erreurs de niveau `E_NOTICE` ont été détectées. La première indique l'oubli des apostrophes dans la définition de la constante `ma_constante`. PHP les a remises, ce qui est correct. La deuxième erreur concerne la variable `$texTe` (avec un « T » majuscule) qui n'est pas définie, d'où l'absence d'affichage. Ce genre de problème survient facilement et est très difficile à détecter. Troisième erreur : on a oublié les

apostrophes dans l'expression `$tableau[second_element]`. PHP n'a pas trouvé de constante nommée `second_element` et suppose donc – à raison – qu'il suffit de remettre les apostrophes. Enfin la dernière erreur est la même que précédemment, mais cette fois la constante existe et PHP la remplace par sa valeur, 5. L'entrée 5 du tableau n'existe pas et un message est donc produit, expliquant l'absence d'affichage pour le dernier élément du tableau.

5.2.2 Gestion des erreurs en PHP

Les erreurs rencontrées ci-dessus sont engendrées par PHP qui se base sur des règles syntaxiques plus ou moins strictes selon le niveau choisi. Ces erreurs sont alors transmises au *gestionnaire d'erreurs* qui détermine comment les traiter. Une erreur PHP est décrite par quatre informations :

1. le niveau d'erreur (voir tableau 5.1) ;
2. le message d'erreur ;
3. le nom du script ;
4. le numéro de la ligne fautive dans le script.

Le gestionnaire d'erreurs par défaut affiche ces informations à l'écran dès que l'erreur survient. On aura donc par exemple :

```
<b>Notice</b>:  Undefined offset:  5 in
        <b>TestErreur.php</b> on line <b>18</b>
```

Ce fonctionnement est très pratique durant la phase de développement d'une application. En plaçant le niveau d'erreur à `E_ALL` (ou même à `E_ALL | E_STRICT` si on développe en PHP 5 « pur »), on affiche tous les messages PHP et on obtient le code le plus propre possible après avoir éliminé leur cause. Ce niveau d'erreur maximal peut être obtenu globalement en modifiant le paramètre `error_reporting` dans le fichier *php.ini*, ou spécifiquement en appelant `error_reporting()` avec la valeur `E_ALL`.

Quand l'application est mise en production, il est plus délicat d'afficher systématiquement des messages qui peuvent correspondre à des erreurs anodines. L'alternative est de rediriger ces messages vers un fichier (*error logging*) en modifiant les paramètres de configuration suivants dans le fichier *php.ini* :

- `display_errors` passe à `Off` ;
- `log_errors` passe à `On` ;
- `error_log` passe à `stderr` ou au nom du fichier de stockage.

Un directive associée, `ignore_repeated_errors`, permet d'éviter (en la positionnant à `On`) la répétition des messages relatifs à une même ligne dans un même fichier. Cela peut servir à ne pas donner l'occasion à un internaute malveillant d'engendrer un très gros fichier par répétition *ad nauseam* de la même manipulation engendrant une erreur.

Quand on utilise Apache, `stderr` est redirigé vers le fichier *error_log*. On peut choisir d'utiliser un fichier comme */tmp/erreurs-php.log*. On y trouvera donc toutes les erreurs engendrées par les applications PHP, qui ne seront plus affichées à l'écran si `display_errors` est positionné à `Off`. Cela suppose bien entendu un suivi régulier de ce fichier pour détecter rapidement les erreurs qui surviennent et ne pas laisser un site « planté » pendant des heures ou des jours.

Signalons que la fonction `error_log()` peut être utilisée d'une part pour écrire directement dans le fichier des erreurs, d'autre part pour être averti par e-mail si on le souhaite. Il semble cependant préférable de mettre en place ce genre de politique grâce aux outils de personnalisation du traitement des erreurs, présentés plus loin, qui offrent l'avantage de pouvoir être redéfinis facilement pour un site particulier, indépendamment du reste de l'application.

Erreurs engendrées par l'application

Bien entendu PHP ne peut pas détecter les erreurs internes correspondant à la rupture de règles propres à l'application. Traditionnellement, on gère ces erreurs tant bien que mal en envoyant un message de détresse à l'écran et en interrompant le script avec `exit` ou `die`. Il est possible de faire mieux en intégrant ces erreurs applicatives dans le système de gestion des erreurs de PHP avec la fonction `trigger_error()` qui prend deux paramètres :

1. le message d'erreur ;
2. le niveau d'erreur parmi `E_USER_NOTICE` (valeur par défaut), `E_USER_WARNING` et `E_USER_ERROR`.

L'utilisation du troisième niveau (`E_USER_ERROR`) provoque de plus l'interruption du script si l'erreur est rencontrée, ce qui revient donc (mais de manière plus propre) à un `exit`. L'avantage de cette solution est que les erreurs sont alors traitées comme des erreurs de syntaxe PHP, ce qui permet de les gérer beaucoup plus souplement en les faisant entrer dans le cadre de la gestion d'erreurs décrite précédemment. Concrètement, on peut, en jouant seulement sur le paramétrage, faire varier le comportement de l'ensemble des scripts en demandant à ce que l'affichage ne se fasse plus à l'écran mais dans un fichier de journalisation, y compris pour les erreurs engendrées par l'application (et gérées explicitement par le programmeur).

La fonction ci-dessous montre quelques exemples d'utilisation de `trigger_error()` pour une fonction de gestion des fichiers transférés d'un client au serveur (voir page 91).

```
function CopieFichierTransmis ($fichier, $destination)
{
  // On récupère le code d'erreur éventuel
  $code_erreur = $fichier['error'];

  if ($code_erreur == UPLOAD_ERR_OK) {
      if (!copy($fichier['tmp_name'], $destination))
        trigger_error("Impossible de copier le fichier !",
                      E_USER_ERROR);
```

```
    }
    else  {
        // Une erreur quelque part?
        switch ($code_erreur)
    {
          case UPLOAD_ERR_INI_SIZE:
            trigger_error("Le fichier dépasse la taille max.
                autorisée par PHP",
                            E_USER_ERROR);
            break;

          case UPLOAD_ERR_FORM_SIZE:
            trigger_error("Le fichier dépasse la taille max. ".
                        "autorisée par le formulaire",
                            E_USER_ERROR);
            break;

          case UPLOAD_ERR_PARTIAL:
            trigger_error("Le fichier a été transféré partiellement
                ",
                            E_USER_ERROR);
            break;
          }
    }
}
```

5.2.3 Les exceptions PHP

Les exceptions existent depuis PHP 5, et sont étroitement associées aux améliorations de la programmation orientée-objet. Le principe des exceptions a été présenté page 124. Rappelons-le brièvement ici, dans une optique de mise en place d'une gestion des erreurs [4].

Les exceptions sont des objets, instanciés par le programmeur, et placés dans un espace réservé de PHP grâce à l'instruction `throw`. Le fait de disposer d'un espace spécifique pour stocker les exceptions évite de les gérer dans la programmation en réservant des variables pour transmettre les codes et les messages d'erreur d'une fonction à l'autre.

On peut, à tout moment, « attraper » les exceptions « lancées » précédemment par un script avec l'instruction `catch`. Comme les erreurs, les exceptions fournissent quatre informations : un message, un code d'erreur (optionnel), le fichier et le numéro de la ligne de l'instruction PHP qui a déclenché l'erreur. Ces informations sont respectivement obtenues par les méthodes `getMessage()`, `getCode()`, `getFile()` et `getLine()` de la classe prédéfinie `Exception`.

4. La discussion qui suit suppose acquises les bases de la programmation objet, telles qu'elles sont présentées dans le chapitre 3.

La classe `Exception` ne demande qu'à être étendue dans des sous-classes personnalisant la gestion des exceptions et la description des erreurs rencontrées. Voici à titre d'exemple une sous-classe `SQLException` destinée à gérer plus précisément les erreurs survenant au cours d'un accès à un SGBD.

Exemple 5.6 *exemples/SQLException.php* : *Extension de la classe* `Exception` *pour les exceptions SQL*

```
<?php
/**
 *  Sous-classe de la classe exception, spécialisée pour
 *  les erreurs soulevées par un SGBD
 */

class SQLException extends Exception
{
  // Propriétés
  private $sgbd; // nom du SGBD utilisé
  private $code_erreur; // code d'erreur du SGBD

  // Constructeur
  function SQLException ($message, $sgbd, $code_erreur=0)
  {
    // Appel du constructeur de la classe parente
    parent::__construct($message);

    // Affectation aux propriétés de la sous-classe
    $this->sgbd = $sgbd;
    $this->code_erreur = $code_erreur;
  }

  // Méthode renvoyant le SGBD qui a levé l'erreur
  public function getSGBD()
  {
    return $this->sgbd;
  }

  // Méthode renvoyant le code d'erreur du SGBD
  public function getCodeErreur()
  {
    return $this->code_erreur;
  }
}
?>
```

On peut alors lancer explicitement une exception instance de `SQLException` et intercepter spécifiquement ce type d'exception. Rappelons encore une fois que toute instance d'une sous-classe est *aussi* instance de toutes les classes parentes, et donc qu'un objet de la classe `SQLException` est aussi un objet de la classe `Exception`, ce qui permet de le faire entrer sans problème dans le moule de gestion des exceptions PHP 5.

Le fragment de code ci-dessous montre comment exploiter cette gestion des exceptions personnalisées.

```
// Bloc d'interception des exceptions
try
{
  // Connexion
  $bd = mysql_connect ((SERVEUR, NOM, PASSE);
  if (!$bd) // Erreur survenue? On lance l'exception
  throw new SQLException ("Erreur de connexion", "MySQL");
  ...
}
catch (SQLException $e) // Interception d'une erreur SQL
{
  trigger_error("Erreur survenue dans " . $e->getSGBD() .
            " : " . $e->getMessage(), E_USER_ERROR);
}
catch (Exception) // Interception de n'importe quelle erreur
{
  trigger_error ("Erreur :  " . $e->getMessage(), E_USER_ERROR);
}
```

On a utilisé plusieurs blocs `catch`, en interceptant les erreurs les plus précises en premier. PHP exécutera le premier bloc `catch` spécifiant une classe dont l'exception est instance.

L'utilisation des exceptions implique leur surveillance et leur interception par une construction `try` et `catch`. Si, quand le script se termine, PHP constate que certaines exceptions n'ont pas été interceptées, il transformera ces exceptions en erreurs standards, avec affichage ou placement dans le fichier des erreurs selon la politique choisie. Le message produit est cependant assez peu sympathique. Voici par exemple ce que l'on obtient si on oublie d'intercepter les exceptions soulevées par la classe d'accès aux bases de données `BD`.

```
Fatal error: Uncaught exception 'Exception' with
   message 'Erreur de connexion au SGBD'
   in BD.class.php:23
```

On peut remplacer ce comportement un peu brutal par un gestionnaire d'exception personnalisé, comme le montrera la prochaine section.

L'introduction des exceptions depuis PHP 5 fait de ce dernier –au moins pour cet aspect – un langage aussi puissant et pratique que C++ ou Java, auxquels il emprunte d'ailleurs très exactement le principe et la syntaxe de cette gestion d'erreurs. Les exceptions offrent un mécanisme natif pour décrire, créer et gérer des erreurs de toutes sortes, sans imposer une gestion « manuelle » basée sur des échanges de codes d'erreur au moment des appels de fonctions, suivi du test systématique de ces codes.

La gestion des exceptions est d'une grande souplesse : on peut spécialiser les différents types d'exception, choisir à chaque instant celle qu'on veut traiter, « relancer »

les autres par un `throw`, séparer clairement les parties relevant de la gestion des erreurs de celles relevant du code de l'application.

Attention cependant : le lancer d'une exception interrompt le script jusqu'au `catch` le plus proche, ce qui n'est pas forcément souhaitable pour toutes les erreurs détectées. Par exemple, quand on teste les données saisies dans un formulaire, on préfère en général afficher d'un coup *toutes* les anomalies détectées pour permettre à l'utilisateur de les corriger en une seule fois. Ce n'est pas possible avec des exceptions.

5.2.4 Gestionnaires d'erreurs et d'exceptions

PHP permet la mise en place de gestionnaires d'erreurs et d'exceptions personnalisés grâce aux fonction `set_error_handler()` et `set_exception_handler()`. Toutes deux prennent en argument une fonction qui implante la gestion personnalisée.

Commençons par la gestion des erreurs. La fonction gestionnaire doit prendre en entrée 5 paramètres : le niveau d'erreur, le message, le nom du script, le numéro de ligne et enfin le contexte (un tableau qui contiendra les variables existantes au moment où la fonction est appelée).

Quand une erreur est déclenchée, par l'interpréteur PHP ou par le développeur via la fonction `trigger_error()`, PHP appelle la fonction gestionnaire d'erreurs en lui passant les valeurs appropriées pour les paramètres. L'exemple ci-dessous montre une fonction de gestion d'erreur.

Exemple 5.7 *webscope/lib/GestionErreurs.php* : *Un gestionnaire d'erreurs PHP*

```
<?php
// Définition d'un gestionnaire d'erreurs.
// Elle affiche le message en français.
function GestionErreurs ($niveau_erreur, $message,
$script, $no_ligne, $contexte=array())
{
  // Regardons le niveau de l'erreur
  switch ($niveau_erreur) {
    // Les erreurs suivantes ne doivent pas être transmises ici!
    case E_ERROR:
    case E_PARSE:
    case E_CORE_ERROR:
    case E_CORE_WARNING:
    case E_COMPILE_ERROR:
    case E_COMPILE_WARNING:
      echo "Cela ne doit jamais arriver !!";
      exit;

    case E_WARNING:
      $typeErreur = "Avertissement PHP";
      break;

    case E_NOTICE:
      $typeErreur = "Remarque PHP";
```

```
        break;

      case E_STRICT:
        $typeErreur = "Syntaxe obsolète PHP 5";
        break;

      case E_USER_ERROR:
        $typeErreur = "Avertissement de l'application";
        break;

      case E_USER_WARNING:
        $typeErreur = "Avertissement de l'application";
        break;

      case E_USER_NOTICE:
        $typeErreur = "Remarque PHP";
        break;

      default:
        $typeErreur = "Erreur inconnue";
    }

    // Maintenant on affiche en rouge
    echo "<font color='red'><b>$typeErreur</b> : " . $message
    . "<br/>Ligne $no_ligne du script $script</font><br/>";

    // Erreur utilisateur? On stoppe le script.
    if ($niveau_erreur == E_USER_ERROR) exit;
}
?>
```

On peut noter que les niveaux d'erreur `E_ERROR`, `E_PARSE`, `E_CORE_ERROR`, `E_CORE_WARNING`, `E_COMPILE_ERROR`, `E_COMPILE_WARNING` sont traités de manière rapide : en principe PHP gèrera *toujours* ce type d'erreur lui-même, sans faire appel au gestionnaire d'erreur ; on ne devrait donc pas les rencontrer ici.

Pour les autres niveaux d'erreur on met en place une gestion personnalisée, consistant ici simplement à afficher les informations en rouge. On peut faire exactement ce que l'on veut : écrire dans un fichier de *log* (journalisation), envoyer un e-mail à l'administrateur, ou toute combinaison appropriée de ces solutions. Une possibilité par exemple est d'une part d'afficher un message neutre et poli à l'utilisateur du site l'informant que l'application est provisoirement indisponible et que l'équipe d'ingénieurs s'active à tout réparer, d'autre part d'envoyer un e-mail à cette dernière pour la prévenir du problème.

Le gestionnaire d'erreurs est mis en place grâce à l'appel suivant :

```
// Gestionnaire personnalisé d'erreurs. Voir GestionErreurs.php.
set_error_handler("GestionErreurs");
```

Soulignons que ceci vient remplacer la gestion normale des erreurs PHP, et qu'il est donc de sa responsabilité d'agir en fonction du niveau détecté. Notre gestionnaire interrompt donc le script avec une instruction `exit` quand une erreur de niveau `E_USER_ERROR` est rencontrée. Si l'on souhaite dans un script abandonner, temporairement ou définitivement, la gestion personnalisée des erreurs, on peut revenir au gestionnaire par défaut avec la fonction `restore_error_handler()`. Cela peut être utile par exemple quand on inclut des scripts PHP pas entièrement compatibles PHP 5 et pour lesquels l'interpréteur engendre des messages d'avertissement.

Le gestionnaire d'exception est basé sur le même principe que le gestionnaire d'erreur : on définit une fonction personnalisée qui prend en entrée un objet instance de la classe `Exception` (et donc de n'importe laquelle de ses sous-classes). Pour faire simple, on peut transformer l'exception en erreur en appelant le gestionnaire d'erreurs défini précédemment.

Exemple 5.8 *webscope/lib/GestionExceptions.php* : *Un gestionnaire d'exceptions PHP*

```
<?php
// Définition d'un gestionnaire d'exceptions. On fait
// simplement appel au gestionnaire d'erreurs

function GestionExceptions ($exception)
{
  // On transforme donc l'exception en erreur
  GestionErreurs (E_USER_ERROR,
                  $exception->getMessage(),
                  $exception->getFile(),
                  $exception->getLine());
}
?>
```

On peut alors mettre en œuvre le gestionnaire d'exceptions grâce à l'appel suivant :

```
set_exception_handler("GestionExceptions");
```

La fonction `GestionExceptions()` sera appelée pour toute exception lancée dans un script qui n'est pas interceptée par un bloc `catch`. Une solution possible est donc de ne pas utiliser du tout les `try` et les `catch` et de se reposer entièrement sur le gestionnaire d'exceptions. C'est d'ailleurs la solution adoptée pour notre site.

Attention à ne pas entrer dans une boucle sans fin en utilisant un gestionnaire d'erreurs qui lance une exception, laquelle à son tour se transforme en erreur et ainsi de suite.

Une fois ces gestionnaires en place, il suffit de les modifier selon les besoins pour obtenir une politique de gestion des erreurs flexible et évolutive.

5.3 PORTABILITÉ MULTI-SGBD

Nous abordons maintenant la question de la portabilité sur plusieurs systèmes de bases de données. Le présent livre est principalement orienté vers MySQL, mais ce produit lui-même s'attache à respecter la norme SQL, ce qui ouvre la perspective de pouvoir porter une application sur d'autres SGBD. Pour un site spécifique, installé en un seul exemplaire, avec le choix définitif d'utiliser MySQL, la question de la portabilité ne se pose pas. À l'autre extrême un logiciel généraliste que l'on souhaite diffuser le plus largement possible gagnera à être compatible avec des systèmes répandus comme PostgreSQL, ORACLE, voire SQLite. SQLite est une interface SQL pour stocker et rechercher des données dans un fichier, sans passer par un serveur de bases de données. SQLite est fourni avec PHP 5 et ne nécessite donc aucune installation autre que celle de PHP.

Le site WEBSCOPE est conçu (et testé) pour être portable, ce qui impose quelques précautions initiales discutées ici.

MySQL est un SGBD relationnel. Il appartient à une famille de systèmes très répandus – ORACLE, PostgreSQL, SQL Server, SYBASE, DB2, le récent SQLite – qui tous s'appuient sur le modèle relationnel de représentation et d'interrogation des données, modèle dont la principale concrétisation est le langage SQL.

En théorie, toute application s'appuyant sur un SGBD relationnel est portable sur les autres. En pratique, chaque système ajoute à la norme SQL ses propres spécificités, ce qui nécessite, quand on veut concevoir une application réellement portable, de bien distinguer ce qui relève de la norme et ce qui relève des extensions propriétaires. Cette section décrit les écueils à éviter et donne quelques recommandations. Le site proposé dans les chapitres qui suivent s'appuie sur ces recommandations pour proposer un code entièrement portable. La seule modification à effectuer pour passer d'un système à un autre est un simple changement de paramètre de configuration. Comme nous allons le voir, le développement d'une application portable n'est pas plus difficile que celle d'une application dédiée, à condition de mettre en place quelques précautions initiales simples.

Cette section peut être omise sans dommage par ceux qui n'envisagent pas d'utiliser un autre système que MySQL.

5.3.1 Précautions syntaxiques

Il faut bien distinguer deux parties dans SQL. Le langage de *définition* de données, ou LDD, permet de créer tous les composants du schéma – principalement les tables. Les commandes sont les `CREATE`, `ALTER`, et `DROP`. Le langage de *manipulation* de données (LMD) comprend les commandes `SELECT`, `UPDATE`, `INSERT` et `DELETE`.

MySQL est très proche de la norme SQL, et tout ce que nous avons présenté jusqu'ici, à quelques exceptions près, relève de cette norme et peut fonctionner sous un autre SGBD. Ces exceptions sont :

1. certains types de données, dont, principalement, `TEXT` ;

2. des constructions comme `ENUM` et `SET` ;
3. l'auto-incrémentation des clés (option `AUTO_INCREMENT` de `CREATE TABLE`).

Il suffit d'ignorer `ENUM` et `SET`. Pour les types de données, MySQL propose un ensemble plus riche que celui de la norme SQL. Le tableau 2.1, page 462, donne la liste des types disponibles et précise ceux qui appartiennent à la norme SQL ANSI : il faut se limiter à ces derniers pour une application portable.

Cela étant, certains types très pratiques, comme `TEXT`, ne sont pas normalisés (ou, plus précisément, la norme SQL qui préconise `BIT VARYING` n'est pas suivie). Il est souvent nécessaire d'utiliser ce type car les attributs de type `VARCHAR` sont limités à une taille maximale de 255 caractères. Le type `TEXT` existe dans PostgreSQL, mais pas dans ORACLE où son équivalent est le type `LONG`. Le script de création de notre schéma, *Films.sql*, page 202, est entièrement compatible avec la norme, à l'exception du résumé du film, de type `TEXT`, qu'il faut donc remplacer par `LONG` si l'on souhaite utiliser ORACLE. Ce genre de modification affecte l'installation, et pas l'utilisation du site, ce qui limite les inconvénients.

Un type normalisé en SQL, mais assez difficile d'utilisation est le type `DATE`. Dans le cadre d'une application PHP, le plus simple est de stocker les dates au format dit « Unix », soit un entier représentant le nombre de secondes depuis le premier janvier 1970. Des fonctions PHP (notamment `getDate()`) permettent ensuite de manipuler et d'afficher cette valeur à volonté.

Pour les mots-clés de SQL et les identificateurs, il n'y a pas de problème si on se limite aux caractères ASCII (mieux vaut éviter les lettres accentuées). L'utilisation des majuscules et minuscules est en revanche un point délicat. Les mots-clés SQL ne sont pas sensibles à la casse, et il en va de même des identificateurs. Pour un système relationnel, toutes les syntaxes suivantes seront donc acceptées, quelle que soit la casse employée pour créer le schéma :

- `SELECT TITRE FROM FILM;`
- `select titre from film;`
- `Select Titre From Film.`

Attention cependant à MySQL qui stocke chaque table dans un fichier dont le nom est celui donné à la table dans la commande `CREATE TABLE`. Sous un système UNIX où les noms de fichiers sont sensibles à la casse, MySQL ne trouvera ni la table *FILM* ni la table *film* si le fichier s'appelle *Film*. Il faut donc toujours nommer les tables de la même manière dans la clause `FROM`, ce qui est facilité par l'emploi d'une convention uniforme comme – par exemple – une majuscule pour la première lettre et des minuscules ensuite.

Il faut de plus prendre en compte PHP qui, lui, est sensible à la casse dans les noms des variables. Les variables `$TITRE`, `$titre` et `$Titre` sont donc considérées comme différentes. Ces noms de variables sont attribués automatiquement par les fonctions PHP d'accès aux bases de données comme `mysql_fetch_object()` (MySQL), `pg_fetch_object()` (PostgreSQL), `oci_fetch_object()` (ORACLE), etc. Tout

dépend de la manière dont ces fonctions nomment les attributs dans les résultats. Or les systèmes appliquent des règles très différentes :

- MySQL utilise la même casse que celle de la clause `SELECT` : après un `SELECT Titre FROM Film` on récupèrera donc une variable `$Titre` ;
- PostgreSQL utilise toujours les minuscules, quelle que soit la casse employée : après un `SELECT Titre FROM Film` on récupèrera donc une variable `$titre` ;
- ORACLE utilise toujours les majuscules, quelle que soit la casse employée : après un `SELECT Titre FROM Film` on récupèrera donc une variable `$TITRE`.

Ces différentes conventions sont dangereuses car elle influent directement sur la correction du code PHP. Avec l'apparition de la couche PDO qui uniformise l'accès aux bases de données depuis la version PHP 5.1, le problème est plus facile à résoudre, mais il est préférable dès le départ d'adopter des noms dattributs loù la casse n'est pas significative : nous avons choisi d'utiliser uniquement les minuscules.

Dernier point auquel il faut être attentif : l'échappement des chaînes de caractères pour traiter les caractères gênants (typiquement, les apostrophes) avant une insertion ou une mise à jour. On utilise traditionnellement la fonction `addSlashes()` qui convient pour MySQL et PostgreSQL, mais par pour ORACLE, SQLite ou SYBASE qui utilisent le doublement des apostrophes. Il faut donc encapsuler la technique d'échappement dans une fonction qui se charge d'appliquer la méthode appropriée en fonction du SGBD utilisé (c'est la méthode `prepareChaine()` de notre classe `BD`).

5.3.2 Le problème des séquences

Voyons maintenant le problème de l'incrémentation automatique des identifiants. Il est très fréquent d'utiliser comme clé primaire d'une table un numéro qui doit donc être incrémenté chaque fois que l'on insère une nouvelle ligne. En l'absence d'un mécanisme spécifique pour gérer ce numéro, on peut penser à prendre le numéro maximal existant et à lui ajouter 1. En SQL cela s'exprime facilement comme ceci :

```
SELECT MAX(id) + 1 FROM <table>
— puis insertion dans la table avec le numéro obtenu
```

Cette solution n'est pas très satisfaisante. Il faut en effet s'assurer que deux sessions utilisateur ne vont pas simultanément effectuer la requête donnant le nouvel id, sous peine de se retrouver avec deux commandes `INSERT` utilisant le même identifiant. On peut verrouiller la table avant d'effectuer la requête `SELECT`, au prix d'un blocage temporaire mais général, y compris pour les sessions qui ne cherchent pas à créer d'identifiant. Enfin, dernier inconvénient, cela peut soulever des problèmes de performances.

Tous les systèmes fournissent donc des générateurs d'identifiants, ou *séquences*. Malheureusement aucun n'applique la même méthode. Dans MySQL, on peut associer une option `AUTO_INCREMENT` à une clé primaire (voir par exemple page 199).

Si on n'indique pas cette clé dans une commande `INSERT`, MySQL se charge automatiquement d'attribuer un nouvel identifiant. De plus il est possible de récupérer l'identifiant précédemment attribué avec la fonction `last_insert_id()`. SQLite emploie la même méthode, sans spécifier `AUTO_INCREMENT`.

Sous ORACLE et PostgreSQL, on utilise des *séquences* [5] qui sont des composants du schéma dédiés à la génération d'identifiants. Une séquence est créée par la commande DDL suivante :

```
CREATE SEQUENCE <nomSéquence>;
```

Il existe, pour chaque système, de nombreuses options permettant d'indiquer la valeur initiale, la valeur maximale, le pas d'incrémentation, etc. Sous PostgreSQL, on peut obtenir un nouvel identifiant en appliquant la fonction `NEXTVAL()` à la séquence. Ensuite, dans la même session, on obtient l'identifiant qui vient d'être attribué avec la fonction `CURRVAL()`. Voici un exemple de session sous PostgreSQL. On crée la séquence, on appelle deux fois `NEXTVAL()` puis une fois `CURRVAL()`.

```
Films=# CREATE SEQUENCE ma_sequence;
CREATE SEQUENCE
Films=# SELECT NEXTVAL('ma_sequence');
 nextval
---------
       1

Films=# SELECT NEXTVAL('ma_sequence');
 nextval
---------
       2

Films=# SELECT CURRVAL('ma_sequence');
 currval
---------
       2
```

Le fonctionnement est pratiquement identique sous ORACLE. Pour obtenir, dans une application PHP, un générateur d'identifiants qui fonctionne sur tous les SGBD, il faut donc écrire une fonction (ou une méthode dans une classe) qui fait appel, selon le système utilisé, à la méthode appropriée. En ce qui concerne MySQL, si on souhaite que l'application soit portable, on ne peut pas utiliser l'auto-incrémentation des lignes de la table ; il faut donc se ramener aux séquences trouvées dans les autres systèmes. On y arrive aisément en créant une table spéciale, avec un seul attribut auto-incrémenté. Chaque insertion dans cette table génère un nouvel identifiant que l'on peut alors obtenir avec la fonction `last_insert_id()`. Voici, sous MySQL, une session équivalente à celle de PostgreSQL.

5. PostgreSQL fournit également un type non standard `SERIAL` qui fonctionne comme l'auto-incrémentation de MySQL.

```
mysql> CREATE TABLE SequenceArtiste
 ->     (id INTEGER NOT NULL AUTO_INCREMENT,
 ->             PRIMARY KEY (id));
mysql>
mysql> insert into SequenceArtiste values();
Query OK, 1 row affected (0,01 sec)

mysql> insert into SequenceArtiste values();
Query OK, 1 row affected (0,00 sec)

mysql> select last_insert_id();
+------------------+
| last_insert_id() |
+------------------+
|                2 |
+------------------+
```

La classe `BD`, décrite dans le chapitre 3, est enrichie d'une méthode abstraite `genereID()`, déclarée comme suit :

```
// Génération d'un identifiant
abstract public function genereID($nomSequence);
```

Cette méthode est ensuite déclinée dans chaque sous-classe correspondant à chaque système. Voici la méthode pour MySQL.

```
// Génération d'un identifiant
public function genereID($nomSequence)
{
  // Insertion d'un ligne pour obtenir l'auto-incrémentation
  $this->execRequete("INSERT INTO $nomSequence VALUES()");

  // Si quelque chose s'est mal passé, on a levé une exception,
  // sinon on retourne l'identifiant
  return mysql_insert_id();
}
```

Et la voici pour PostgreSQL.

```
// Génération d'un identifiant
public function genereID($nomSequence)
{
  // Appel à la séquence
  $res = $this->execRequete("SELECT NextVal('$nomSequence') AS id");
  $seq = $this->objetSuivant($res);
  return $seq->id;
}
```

La gestion des séquences est le seul aspect pour lequel la programmation d'une application PHP/MySQL s'écarte légèrement des techniques que l'on emploierait si

on ne visait pas une application portable. Comme on le voit avec la solution adoptée ci-dessus, la modification est d'une part tout à fait mineure, d'autre part invisible pour l'application qui se contente d'appeler le générateur quand elle en a besoin.

5.3.3 PDO, l'interface générique d'accès aux bases relationnelles

La dernière chose à faire pour assurer la portabilité de l'application est d'utiliser une interface normalisée d'accès à la base de données, qui cache les détails des API propres à chaque système, comme le nom des fonctions, l'ordre des paramètres, le type du résultat, etc. Depuis la version 5.1 de PHP cette interface existe de manière standardisée sour le nom *PHP Data Objects* (PDO). PDO ne dispense pas des précautions syntaxiques présentées ci-dessus, mais fournit des méthodes d'accès standardisées à une base de données, quel que soit le système sous-jacent.

PDO ne présente aucune difficulté maintenant que vous êtes rôdés à l'interface PHP/MySQL. Voici un exemple similaire au script *ApplClasseMySQL.php*, page 119, pour interroger la table *FilmSimple*.

Exemple 5.9 *exemples/AppIPDO.php* : *Utilisation de PDO*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<head>
<title>Interface PDO</title>
<link rel='stylesheet' href="films.css" type="text/css" />
</head>
<body>

<h1>Illustration de l'interface PDO</h1>

<?php

/**
 * Exemple de programmation avec PDO
 */

require_once ("Connect.php");

try {
  // On se connecte
  $bd = new PDO('mysql:host='.SERVEUR.';dbname='.BASE, NOM, PASSE
      );

  // On exécute une requête
  $resultat = $bd->query ("SELECT * FROM FilmSimple");

  // On récupère les lignes
  while ($film = $resultat->fetch (PDO::FETCH_OBJ)) {
```

```
    echo "<b>$film->titre</b>, paru en $film->annee, réalisé "
    . "par $film->prenom_realisateur $film->nom_realisateur.<br
      />\n";
  }

  // Et on ferme le curseur
  $resultats->closeCursor();
}
catch(Exception $e) {
  echo 'Erreur PDO : ' . $e->getCode . " -- " . $e->getMessage()
    . '<br />';
}

?>
</body>
</html>
```

On commence donc par instancier une connexion avec la base de données. Il s'agit d'un objet de la classe `PDO`, dont le constructeur prend en entrée les paramètres habituels : serveur, nom de la base, et compte de connexion. On précise également que l'on se connecte à MySQL. C'est le seul point à modifier pour utiliser un autre système.

On peut ensuite exécuter une requête d'interrogation avec la méthode `query()`. Elle renvoie un objet instance de `PDOStatement` qui sert à parcourir le résultat avec la méthode `fetch()`. On passe à cette dernière méthode le format (objet ou tableau) dans lequel on souhaite obtenir le résultat.

Tout est donc semblable, à quelques détails près, à ce que nous utilisons depuis plusieurs chapitres pour MySQL. Quand on veut protéger par un échappement les données à insérer dans une requête, on utilise la méthode `quote()`. Notez également que PDO distingue les requêtes d'interrogation, exécutées avec `query()`, des requêtes de mise à jour pour lesquelles on utilise `exec()`.

Si vous voulez créer une application portable multi-SGBD, l'apprentissage de PDO ne pose aucun problème. Nous y revenons de manière plus complète dans le cadre de la programmation avec le Zend Framework, chapitre 9. Pour le site WEB-SCOPE, nous continuons à utiliser la classe abstraite `BD`, conçue dans le même but, et dont la réalisation est décrite dans le chapitre 3. Rappelons que cette classe fixe les méthodes communes à tous les systèmes, et se décline en sous-classes implantant ces méthodes pour chaque système utilisé. Rien n'empêche de revoir l'implantation de cette classe avec PDO, de manière transparente pour le reste de l'application. Nous pouvons donc considérer que notre application est portable d'un SGBD à un autre.

6

Architecture du site : le *pattern* MVC

Ce chapitre est consacré au « motif de conception » (*design pattern*) *Modèle-Vue-Contrôleur* (MVC). Ce *pattern* est maintenant très répandu, notamment pour la réalisation de sites web, et mène à une organisation rigoureuse et logique du code.

Un des objectifs est la séparation des différentes « couches » constituant une application, de manière à pouvoir travailler indépendamment sur chacune. Il devrait par exemple toujours être possible de revoir complètement la présentation d'un site sans toucher au code PHP, et, réciproquement, le code PHP devrait être réalisé avec le minimum de présupposés sur la présentation. La question de l'évolutivité du code est elle aussi essentielle. Un logiciel évolue toujours, et doit donc être modifiable facilement et sans dégradation des fonctions existantes (régression). Enfin, dans tous les cas, l'organisation du code doit être suffisamment claire pour qu'il soit possible de retrouver très rapidement la partie de l'application à modifier, sans devoir ouvrir des dizaines de fichiers.

Ce chapitre présente le MVC dans un contexte pratique, en illustrant les différentes composantes par des fonctionnalités intégrées au site WEBSCOPE. De fait, à la fin du chapitre nous disposerons d'un cadre de développement MVC dans lequel l'ensemble du site prendra place. Pour des raisons de clarté et d'introduction à des concepts parfois complexes, le MVC présenté ici vise davantage à la simplicité et à la légèreté qu'à la richesse. L'apprentissage de solutions plus complètes destinées à des développements à grande échelle devrait en être facilité. J'espère vous convaincre ainsi de l'intérêt de cette approche pour toutes vos réalisations.

6.1 LE MOTIF DE CONCEPTION MVC

Cette introduction au MVC est volontairement courte afin de dire l'essentiel sans vous surcharger avec toutes les subtilités conceptuelles qui accompagnent le sujet. Je passe ensuite directement aux aspects pratiques avec la réalisation « maison » du MVC que nous allons utiliser pour implanter notre site.

6.1.1 Vue d'ensemble

L'objectif global du MVC est de séparer les aspects *traitement*, *données* et *présentation*, et de définir les interactions entre ces trois aspects. En simplifiant, les données sont gérées par le *modèle*, la présentation par la *vue*, les traitements par des *actions* et l'ensemble est coordonné par les *contrôleurs*. La figure 6.1 donne un aperçu de l'architecture obtenue, en nous plaçant d'emblée dans le cadre spécifique d'une application web.

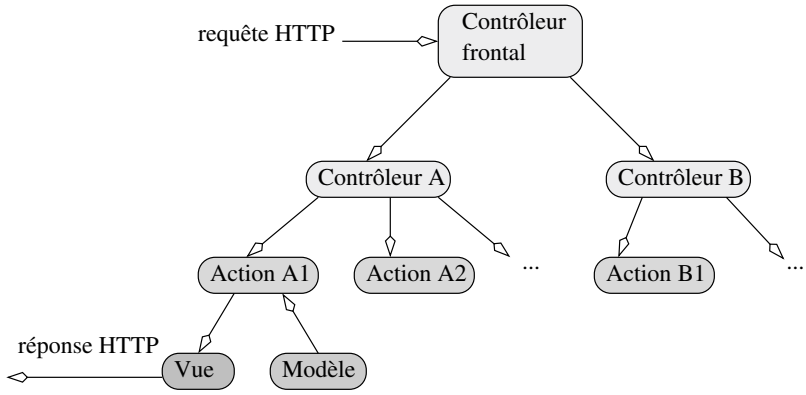


Figure 6.1 — Aperçu général d'une application MVC

La figure montre une application constituée de plusieurs contrôleurs, chacun constitué d'un ensemble d'actions. La première caratéristique de cette organisation est donc de structurer hiérarchiquement une application. Dans les cas simples, un seul contrôleur suffit, contenant l'ensemble des actions qui constituent l'application. Pour de très larges applications, on peut envisager d'ajouter un niveau, les *modules*, qui regroupent plusieurs contrôleurs.

Chaque requête HTTP est prise en charge par une action dans un contrôleur. Il existe un *contrôleur frontal* qui analyse une requête HTTP, détermine cette action et se charge de l'exécuter en lui passant les paramètres HTTP.

Au niveau du déroulement d'une action, les deux autres composants, la vue et le modèle, entrent en jeu. Dans le schéma de la figure 6.1, l'action A_1 s'adresse au modèle pour récupérer des données et peut-être déclencher des traitements spécifiques à ces données. L'action passe ensuite les informations à présenter à la vue qui se charge de créer l'affichage. Concrètement, cette présentation est le plus souvent un document HTML qui constitue la réponse HTTP.

Il s'agit d'un schéma général qui peut se raffiner de plusieurs manières, et donne lieu à plusieurs variantes, notamment sur les rôles respectifs des composants. Sans entrer dans des discussions qui dépassent le cadre de ce livre, voici quelques détails sur le modèle, la vue et le contrôleur.

6.1.2 Le modèle

Le modèle est responsable de la préservation de l'*état* d'une application entre deux requêtes HTTP, ainsi que des fonctionnalités qui s'appliquent à cet état. Toute donnée persistante doit être gérée par la couche « modèle ». Cela concerne les données de session (le panier dans un site de commerce électronique par exemple) ou les informations contenues dans la base de données (le catalogue des produits en vente, pour rester dans le même exemple). Cela comprend également les règles, contraintes et traitements qui s'appliquent à ces données, souvent désignées collectivement par l'expression « logique de l'application ».

6.1.3 La vue

La vue est responsable de l'interface, ce qui recouvre essentiellement les fragments HTML assemblés pour constituer les pages du site. Elle est également responsable de la mise en forme des données (pour formater une date par exemple) et doit d'ailleurs se limiter à cette tâche. Il faut prendre garde à éviter d'introduire des traitements complexes dans la vue, même si la distinction est parfois difficile. En principe la vue ne devrait pas accéder au modèle et obtenir ses données uniquement de l'action (mais il s'agit d'une variante possible du MVC).

La vue est souvent implantée par un moteur de *templates* (que l'on peut traduire par « gabarit »), dont les caractéristiques, avantages et inconvénients donnent lieu à de nombreux débats. Nous utiliserons un de ces moteurs dans notre MVC, ce qui vous permettra de vous former votre propre opinion.

6.1.4 Contrôleurs et actions

Le rôle des contrôleurs est de récupérer les données utilisateur, de les filtrer et les contrôler, de déclencher le traitement approprié (via le modèle), et finalement de déléguer la production du document de sortie à la vue. Comme nous l'avons indiqué précédemment, l'utilisation de contrôleurs a également pour effet de donner une structure hiérarchique à l'application, ce qui facilite la compréhension du code et l'accès rapide aux parties à modifier. Indirectement, la structuration « logique » d'une application MVC en contrôleurs et actions induit une organisation physique adaptée.

6.1.5 Organisation du code et conventions

La figure 6.2 montre les répertoires constituant l'organisation du code de notre application WEBSCOPE.

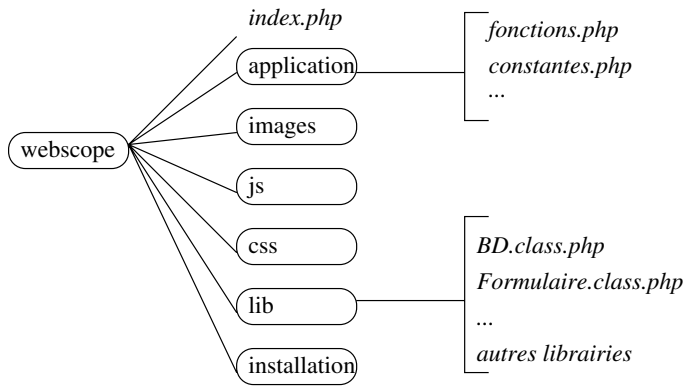


Figure 6.2 — Organisation du code

Première remarque importante : *toutes* les requêtes HTTP sont traitées par un unique fichier *index.php*. Ce choix permet de rassembler dans un seul script les inclusions de fichiers, initialisations et réglages de configuration qui déterminent le contexte d'exécution de l'application. Toutes les URL de l'application sont de la forme :

http://serveur/webscope/index.php?ctrl=`nomctrl`*&action=*`nomact`*[autres paramètres]*

On indique donc, avec des paramètres HTTP (ici en mode `get`), le nom du contrôleur `nomctrl` et le nom de l'action `nomact`. Ces paramètres sont optionnels : par défaut le nom du contrôleur est `Index` et le nom de l'action est `index` (notez que, par convention, les contrôleurs commencent par une majuscule, et pas les actions). La requête HTTP:

http://serveur/webscope/index.php

déclenche donc l'action `index` du contrôleur `Index`. On peut même omettre l'indication du script *index.php* si le serveur web utilise ce script par défaut.

> **REMARQUE –** Il faudrait mettre en place un mécanisme pour s'assurer que toute URL incorrecte est redirigée vers *index.php* ; il faudrait aussi, pour des raisons de sécurité, placer tous les fichiers qui ne peuvent pas être référencés directement dans une URL (par exemple les classes et bibliothèques de *lib*) en dehors du site web. Voir le chapitre 9 pour ces compléments.

Revenons à l'organisation du code de la figure 6.2. Les répertoires *css*, *images* et *js* contiennent respectivement les feuilles de style CSS, les images et les scripts Javascript. Le répertoire *installation* contient les fichiers permettant la mise en route de l'application (par exemple des scripts SQL de création de la base). Les deux répertoires *lib* et *application* sont plus importants.

- *lib* contient tous les utilitaires indépendants des fonctionnalités de l'application (le code « structurel ») : connexion et accès aux bases de données ;

production de formulaires ; classes génériques MVC, bibliothèques externes pour la production de graphiques, l'accès à des serveurs LDAP, etc.

- *application* contient tout le code fonctionnel de l'application : les contrôleurs (répertoire *controleurs*), les modèles (répertoire *modeles*), les vues (répertoire *vues*), les fonctions et les classes, etc.

Placer indépendamment les bibliothèques et utilitaires permet une mise à jour plus facile quand de nouvelles versions sont publiées. D'une manière générale, cette organisation permet de localiser plus rapidement un fichier ou une fonctionnalité donnée. C'est une version un peu simplifiée des hiérarchies de répertoires préconisées par les *frameworks*, que nous étudierons dans le chapitre 9.

À cette organisation s'ajoutent des conventions d'écriture qui clarifient le code. Celles utilisées dans notre site sont conformes aux usages les plus répandus :

1. les noms de classes et de fonctions sont constitués d'une liste de mots-clés, chacun commençant par une majuscule (exemple : `AfficherListeFilms()`) ;
2. les noms de méthodes suivent la même convention, à ceci près que la première lettre est une minuscule (exemple : `chercheFilm()`) ;
3. les noms de tables, d'attributs et de variables sont en minuscules ; (exemple : `date_de_naissance`) ;
4. les constantes sont en majuscules (exemple : `SERVEUR`) ;
5. les contrôleurs s'appelent *`nom`*`Ctrl`, et sont des classes héritant de la classe `Controleur` (exemple : `SaisieCtrl()`) ; les actions d'un contrôleur sont les méthodes de la classe.

On distingue ainsi du premier coup d'œil, en regardant un script, les différentes catégories syntaxiques. Tous ces choix initiaux facilitent considérablement le développement et la maintenance.

6.2 STRUCTURE D'UNE APPLICATION MVC : CONTRÔLEURS ET ACTIONS

Voyons maintenant le fonctionnement des contrôleurs et la manière dont l'application détermine l'action à exécuter.

6.2.1 Le fichier *index.php*

Commençons par le script *index.php*, ci-dessous.

Exemple 6.1 *webscope/index.php* : *L'unique script recevant des requêtes HTTP*

```
<?php
// Indique le niveau des erreurs
error_reporting(E_ALL | ~E_STRICT);
```

```
// Zone par défaut pour les calculs de date
date_default_timezone_set("Europe/Paris");

// Calcule automatiquement le chemin depuis la racine
// jusqu'au répertoire courant
$root = dirname(__FILE__) . DIRECTORY_SEPARATOR ;

// On complète la liste des chemins d'inclusion
set_include_path('.' .
    PATH_SEPARATOR . $root . 'lib' . DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . $root . 'application' . DIRECTORY_SEPARATOR
        .
    PATH_SEPARATOR . $root . 'application/modeles' .
        DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . $root . 'application/fonctions' .
        DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . $root . 'application/classes' .
        DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . get_include_path()
);

// La configuration
require_once ("Config.php");

// Classes de la bibliothèque
require_once ("Tableau.php");
require_once ("Formulaire.php");
require_once ("BDMySQL.php");
require_once ("Template.php");

// Fonctions diverses
require_once ("NormalisationHTTP.php");
require_once ("GestionErreurs.php");
require_once ("GestionExceptions.php");

// Si on est en échappement automatique, on annule
// les échappements pour que l'application soit indépendante
// de magic_quote_gpc
if (get_magic_quotes_gpc()) {
  $_POST = NormalisationHTTP($_POST);
  $_GET = NormalisationHTTP($_GET);
  $_REQUEST = NormalisationHTTP($_REQUEST);
  $_COOKIE = NormalisationHTTP($_COOKIE);
}

// Indiquons si on affiche ou pas les erreurs, avec
// la constante venant de Config.php
ini_set("display_errors", DISPLAY_ERRORS );

// Gestionnaire personnalisé d'erreurs. Voir GestionErreurs.php.
set_error_handler("GestionErreurs");
```

```
// Gestionnaire personnalisé d'exceptions. Voir GestionExceptions
// .php.
set_exception_handler("GestionExceptions");

// Tout est prêt. On charge le contrôleur frontal
require_once('Frontal.php');
$frontal = new Frontal();

// On demande au contrôleur frontal de traiter la requête HTTP
try {
  $frontal->execute();
}
catch (Exception $e) {
  echo "Exception levée dans l'application. <br/>"
       . "<b>Message</b> " . $e->getMessage() . "<br/>"
       . "<b>Fichier</b> " . $e->getFile()
       . "<b>Ligne</b> " . $e->getLine() . "<br/>";
}
```

Les commentaires indiquent assez clairement le rôle de chaque instruction. L'appel de la fonction `set_include_path()` est étroitement lié à l'organisation du code : il permet de placer dans la liste des répertoires d'inclusion de l'interpréteur PHP ceux qui contiennent nos fonctions et classes.

```
set_include_path('.' .
    PATH_SEPARATOR . $root . 'lib' . DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . $root . 'application' . DIRECTORY_SEPARATOR
        .
    PATH_SEPARATOR . $root . 'application/modeles' .
        DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . $root . 'application/fonctions' .
        DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . $root . 'application/classes' .
        DIRECTORY_SEPARATOR .
    PATH_SEPARATOR . get_include_path()
);
```

Les constantes `PATH_SEPARATOR` et `DIRECTORY_SEPARATOR` sont définies par PHP et servent à ne pas dépendre du système hôte (Linux ou Mac OS X utilisent le « / » pour séparer les noms de répertoire, Windows le « \ »).

On peut ensuite charger les utilitaires nécessaires au fonctionnement de l'application, avec `require_once()`. Notez qu'on n'indique pas le chemin d'accès vers les contrôleurs, car ceux-ci ne sont pas systématiquement chargés. Seul le contrôleur concerné par une requête est chargé, et c'est le contrôleur frontal qui s'en charge avec sa méthode `execute()`.

La directive `magic_quotes_gpc` est susceptible de changer d'une configuration à une autre, passant de `On` à `Off`. Ce problème a été discuté en détail page 68, et la solution préconisée alors est ici appliquée : toutes les données provenant de HTTP

sont « normalisées » pour annuler l'échappement qui a éventuellement été pratiqué suite au paramétrage à `On` de `magic_quotes_gpc`.

On peut ensuite développer tout le reste du site en considérant que `magic_quotes_gpc` vaut `Off`. Il serait bien sûr plus facile de pouvoir changer la configuration au moment de l'exécution mais ce n'est pas possible pour cette directive. Il est probable qu'elle sera supprimée en PHP 6.

Notez enfin qu'on utilise la fonction `init_set()` pour fixer le paramètre `display_errors`. Sa valeur est déterminée par la constante `DISPLAY_ERRORS`, définie dans le fichier *Config.php*, qu'il faut absolument placer à `Off` sur un site en production.

6.2.2 Le contrôleur frontal

Le contrôleur frontal est une instance de la classe `Frontal` dont le rôle est de « router » la requête HTTP vers le contrôleur et l'action appropriés. Comme d'habitude avec l'approche orientée-objet, on peut se contenter d'utiliser une classe sans connaître son implantation, ou inspecter cette dernière pour se faire une idée de la manière dont les choses sont traitées. Pour satisfaire votre curiosité, voici le code de la méthode `execute()` dans *Frontal.php* (ce fichier se trouve dans *lib*).

```
function execute ()
{
  // D'abord, on récupère les noms du contrôleur et de l'action
  if (isSet($_GET['controleur']))
    $controleur = ucfirst($_GET['controleur']) . "Ctrl";
  else
    $controleur = "IndexCtrl";

  if (isSet($_GET['action']))
    $action = lcfirst($_GET['action']);
  else
    $action = "index";

   // Maintenant chargeons la classe
    $chemin = "controleurs" . DIRECTORY_SEPARATOR . $controleur
         . ".php";
   if(file_exists("application" . DIRECTORY_SEPARATOR . $chemin
       )) {
      require_once($chemin);
    } else {
      throw new Exception ("Le contrôleur <b>$controleur</b> n'
          existe pas");
    }

    // On instancie un objet
    eval ("\$ctrl = new $controleur();");

    // Il faut vérifier que l'action existe
    if (!method_exists($ctrl, $action)) {
```

```
            throw new Exception ("L'action <b>$action</b> n'existe pas
                ");
        }

        // Et pour finir il n'y a plus qu'à exécuter l'action
        call_user_func(array($ctrl, $action));
    }
```

Essentiellement, elle détermine le contrôleur et l'action en fonction des paramètres reçus dans la requête HTTP. Conformément à nos conventions, un contrôleur nommé *controL* est implanté par une classe nommée *controL*`Ctrl` et se trouve dans le fichier *controL*`Ctrl.php` du répertoire *application/controleurs*. On vérifie donc que ce fichier existe, faute de quoi on lève une exeption. On instancie ensuite ce contrôleur avec la fonction `eval()`, qui permet d'évaluer une expression PHP construite dynamiquement (ce qui est le cas ici puisqu'on ne sait pas à l'avance quel est le nom de la classe à instancier). Finalement on vérifie que l'action demandée est bien implantée par une méthode dans la classe instanciée, et si oui, on l'exécute.

Ce fragment de code est un exemple de ce que l'on pourrait appeler « métaprogrammation » en PHP : on crée par programmation du code PHP que l'on exécute. C'est une pratique assez courante en programmation avancée, car elle permet de résoudre élégamment des problèmes assez difficiles à traiter dans des languages moins souples.

En résumé, le contrôleur frontal charge la classe du contrôleur, l'instancie et exécute la méthode correspondant à l'action. Voyons maintenant le contrôleur lui-même.

6.2.3 Créer des contrôleurs et des actions

Créer un contrôleur est extrêmement simple : on ajoute un fichier *nom*`Ctrl.php` dans *application/controleurs*, où *nom* est le nom du contrôleur. Ce fichier contient une classe qui hérite de `Controleur`. Voici le code d'un contrôleur servant d'exemple.

Exemple 6.2 *webscope/application/controleurs/TestCtrl.php* : *Le contrôleur* `test`

```
<?php
/**
 * @category webscope
 * @copyright Philippe Rigaux, 2008
 * @license GPL
 * @package test
 */

require_once ("Controleur.php");

/**
 * Controleur de test, montrant comment implanter
 * des actions dans un contrôleur.
 */
```

```
class TestCtrl extends Controleur
{

  /**
   * Action par défaut – affichage de la liste des régions
   */
  function index ()
  {
    // Affichage de la liste des régions
    $resultat = $this->bd->execRequete ("SELECT * FROM Region");
    while ($region = $this->bd->objetSuivant ($resultat))
      echo "<b>$region->nom</b><br/>";
  }
}
?>
```

Le code est une classe qui sert simplement de « coquille » à une liste de méthodes publiques, sans paramètre, implantant les actions. Ajouter une action revient donc à ajouter une méthode. La seule action disponible ici est `index`, que l'on appelle avec l'URL :

http://serveur/webscope/index.php?ctrl=test&action=index

Ou bien, plus simplement

http://serveur/webscope/?ctrl=test

en tirant parti du fait qu'`index` est l'action par défaut, et *index.php* le script par défaut.

En étudiant cette action, on constate que l'objet-contrôleur dispose d'une propriété `$this->bd`, qui permet d'exécuter des requêtes. D'où vient cet objet ? De la super-classe `Controleur` qui instancie automatiquement un objet de la classe `BDMySQL` dans son constructeur. Tous les contrôleurs, sous-classes de `Controleur`, héritent de ce constructeur et, automatiquement, on dispose donc d'une connexion avec la base. Voici le code du constructeur de `Controleur`.

```
function __construct ()
{
  /*
   * Le contrôleur initialise plusieurs objets utilitaires :
   *  – une instance de BD pour accéder à la base de données
   *  – une instance du moteur de templates pour gérer la vue
   */

  // Initialisation de la session PHP
  session_start ();

  // Connexion à la base
  $this->bd = new BDMySQL (NOM, PASSE, BASE, SERVEUR);
```

```
        // Instanciation du moteur de templates
        $this->vue = new Template ("application" .
            DIRECTORY_SEPARATOR . "vues");

        // On charge systématiquement le "layout" du site
        $this->vue->setFile("page", "layout.tpl");

        // et initialisation du contenu et du titre.
        $this->vue->contenu ="";
        $this->vue->titre_page = "";

        // Recherche de la session
        $this->initSession (session_id());

        // Initialisation de la partie du contenu
        // qui montre soit un formulaire, de connexion,
        // soit un lien de déconnexion
        $this->statutConnexion();
    }
```

On peut noter que le constructeur instancie également un moteur de templates pour gérer la vue, accessible dans `$this->vue`, ainsi que des informations relatives à la session. Nous y reviendrons.

Au sein d'une action, on programme en PHP de manière tout à fait classique. Il ne s'agit pas vraiment de programmation orientée-objet au sens où nous l'avons vu dans les chapitres précédents. L'approche objet se borne ici à structurer le code, et à bénéficier du mécanisme d'héritage pour initialiser des composants utiles à toutes les actions.

Retenez cette approche consistant à définir une super-classe pour définir un comportement commun à un ensemble d'objets (ici les contrôleurs). Toutes les tâches répétitives d'intialisation de l'environnement, de configuration, de connexion à la base, etc., sont déjà faites une fois pour toutes. Inversement, cela rend très facile l'ajout de nouvelles contraintes, communes à tous les objets, par enrichissement de la super-classe. Un simple exemple : que se passe-t-il si on écrit un contrôleur en oubliant une méthode nommée `index()` ? Alors le choix par défaut effectué par le contrôleur frontal risque d'entraîner une erreur puisque l'action par défaut, `index`, n'existe pas. Solution : on définit cette action par défaut dans la super-classe `Controleur` : elle existe alors, par héritage, dans *tous* les contrôleurs, et elle est *surchargée* par toute méthode `index()` définie au niveau des sous-classes.

6.3 STRUCTURE D'UNE APPLICATION MVC : LA VUE

Le code de l'action `index()` du contrôleur `test`, présenté précédemment, affiche simplement la sortie avec la commande PHP `echo`. C'est contraire au principe MVC de séparer la production de la présentation du traitement des données. L'inconvénient est de se retrouver à manipuler de très longues chaînes de caractères HTML dans le script, pratique qui mène extrêmement rapidement à un code illisible.

Une solution très simple consisterait à organiser chaque page en trois parties, en-tête, contenu et pied de page, l'en-tête et le pied de page étant systématiquement produits par des fonctions PHP `Entete()` et `PiedDePage()`. La figure 6.3 montre le style d'interaction obtenu, chaque action (sur la gauche) produisant les différentes parties de la page.

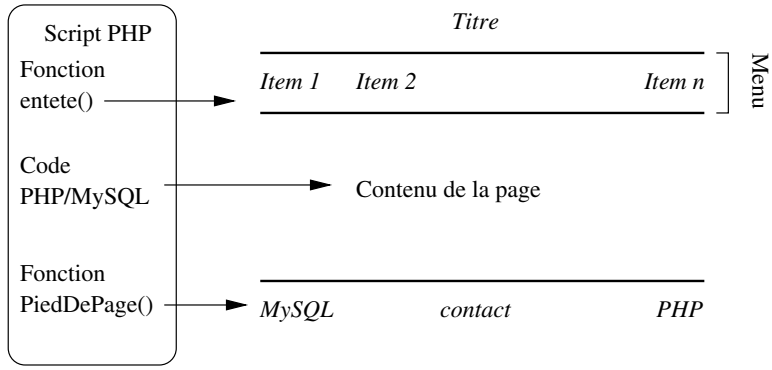


Figure 6.3 — Tout le code HTML est produit avec PHP.

Cette méthode est envisageable pour de petits sites pour lesquels la conception graphique est stable et peu compliquée. Elle offre l'avantage de regrouper en un seul endroit (nos deux fonctions) les choix de présentation, et de rendre l'application indépendante de tout outil de production HTML.

Pour des projets plus conséquents, il nous faut un composant

- gérant la vue,
- offrant une séparation claire entre les fragments HTML constituant la présentation des pages et le code PHP qui fournit le contenu.

L'approche basée sur des *templates*, ou modèles de présentation, dans lesquels on indique les emplacements où le contenu produit dynamiquement doit être inséré, constitue une solution pratiquée depuis très longtemps. Elle offre plusieurs avantages, et quelques inconvénients. Pour être concret, je vais donner des exemples de la gestion de la vue à base de templates, avant de revenir sur les principes généraux de séparation du code HTML et du code PHP.

6.3.1 Les *templates*

Le système utilisé pour nos exemples est un moteur de templates adapté de la bibliothèque PHPLIB et amélioré grâce aux possibilités de PHP 5. Ce moteur est très représentatif des fonctionnalités des *templates* (dont il existe de très nombreux représentants) et s'avère simple à utiliser. Les méthodes publiques de la classe sont données dans le tableau 6.1.

Tableau 6.1 — Méthodes de la classe `Template`

Méthodes	Description
__construct(*racine*)	Constructeur
setFile(*nom*, *fichier*)	Charge un fichier dans une entité nommée *nom*. On peut également passer en premier paramètre un tableau contenant la liste des fichiers à charger.
setBlock(*nom*, *nomBloc*, *nomRemplaçant*)	Remplace, dans le contenu de l'entité *nom*, le bloc *nomBloc* par une référence à l'entité *nomRemplaçant*, et crée une nouvelle entité *nomBloc*.
assign(*nomCible*, *nomSource*)	Place dans *nomCible* le contenu de *nomSource* dans lequel les références aux entités ont été remplacées par leur contenu.
append(*nomCible*, *nomSource*)	Ajoute (par concaténation) à *nomCible* le contenu de *nomSource* dans lequel les références aux entités ont été remplacées par leur contenu.
render(*nomCible*)	Renvoie le contenu de *nomCible*.

Un *template* est un fragment de code HTML (ou tout format textuel) qui fait référence à des *entités*. Une entité est simplement un nom qui définit une association entre le code PHP et la sortie HTML.

1. dans un *template*, on trouve les *références* d'entités, entourées par des accolades ;
2. dans le code PHP, une entité est une variable du composant Vue, à laquelle on affecte une valeur.

Lors de l'exécution, la référence à une entité est substituée par sa valeur, qui peut aussi bien être une simple donnée qu'un fragment HTML complexe. C'est le moteur de templates qui se charge de cette substitution (ou *instanciation*).

Commençons par un exemple simple. Le but est de construire une page en assemblant d'une part un fragment HTML sans aucune trace de PHP, et d'autre part une partie PHP, sans aucune trace de HTML. Le système de *templates* se chargera de faire le lien entre les deux. Voici tout d'abord la partie HTML (l'extension choisie ici est, par convention, `.tpl` pour « template »).

Exemple 6.3 *exemples/ExTemplate.tpl* : *Le fichier modèle*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Exemple de template</title>

<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
```

```
<body>

<!— Exemple simple d'utilisation des templates.
    Notez qu'il n'y a pas une trace de PHP ci—dessous.
 —>

<h1>{titre_page}</h1>

Cette page a été engendrée  par le système de templates.
Elle contient des éléments statiques, comme la phrase
que vous êtes en train de lire. Mais on y trouve
également des parties dynamiques produites avec
PHP, comme le nom de votre navigateur : <b>{nom_navigateur}.</b>
<p>
On peut aussi afficher la date et l'heure:
Nous sommes le <b>{date}</b>, il est <b>{heure}</b> heure(s).
</p>
<p>
Pourtant la personne qui a produit le code HTML
ne connaît rien à PHP, et la personne qui programme en PHP
n'a aucune idée de la mise en forme choisie.
Intéressant non ?
</p>
</body>
</html>
```

C'est donc du HTML standard où certains éléments du texte, les *références d'entités*, désignent les parties dynamiques produites par PHP. Les références d'entités sont encadrées par des accolades, comme `{titre_page}`. Voici maintenant la partie PHP.

Exemple 6.4 *exemples/ExTemplate.php* : *Le fichier PHP*

```
<?php
// Exemple  simple d'utilisation des templates.
// Notez qu'il n'y a pas une trace de HTML ci—dessous.

// Inclusion du moteur de templates
require ("Template.php");

// Instanciation d'un objet de la classe Template
$tpl = new Template (".");

// Chargement dans l'entité 'page' du fichier contenant le
    template
$tpl—>setFile ("page", "ExTemplate.tpl");

// On donne une valeur aux entités référencées
$tpl—>titre_entete = "Les templates";
$tpl—>titre_page =  "Un exemple de templates";
$tpl—>date = date ("d/m/Y");
```

```
$tpl->heure =  date ("H");
$tpl->nom_navigateur = $_SERVER['HTTP_USER_AGENT'];

// La méthode render remplace les références par leur valeur, et
    renvoie
// la nouvelle chaîne de caractères.
echo $tpl->render("page");
?>
```

Le principe est limpide : on crée un objet de la classe *Template* en lui indiquant que les fichiers de modèles sont dans le répertoire courant, « . ». On commence par charger le contenu du fichier *ExTemplate.tpl* et on l'affecte à l'entité `page`, qui contient donc des références à d'autres entités (`date`, `heure`, etc.). Il faut alors donner une valeur à ces entités avec l'opérateur d'affectation '='. Par exemple :

```
$tpl->date = date ("d/m/Y");
$tpl->heure =  date ("H");
```

Maintenant on peut *substituer* aux références d'entité présentes dans `page` les valeurs des entités qu'on vient de définir. Cela se fait en appelant la méthode `render()`. Elle va remplacer `{date}` dans l'entité `page` par sa valeur, et de même pour les autres références. Il ne reste plus qu'à afficher le texte obtenu après substitution. On obtient le résultat de la figure 6.4 qui montre qu'avec très peu d'efforts, on a obtenu une séparation complète de PHP et de HTML.

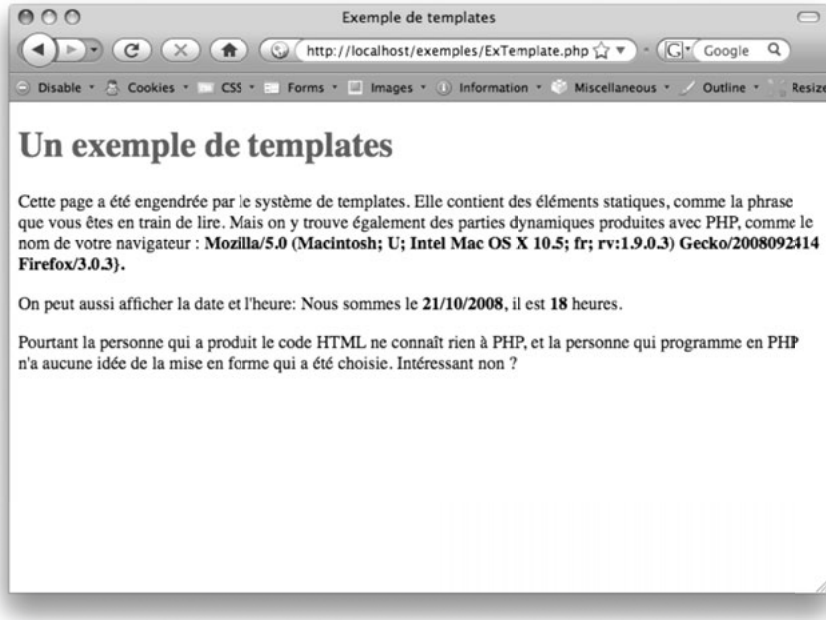

Exemple de templates
http://localhost/exemples/ExTemplate.php
Google
Disable
Cookies
CSS
Forms
Images
Information
Miscellaneous
Outline
Resize
Un exemple de templates
Cette page a été engendrée par le système de templates. Elle contient des éléments statiques, comme la phrase que vous êtes en train de lire. Mais on y trouve également des parties dynamiques produites avec PHP, comme le nom de votre navigateur : Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; fr; rv:1.9.0.3) Gecko/2008092414 Firefox/3.0.3}.
On peut aussi afficher la date et l'heure: Nous sommes le 21/10/2008, il est 18 heures.
Pourtant la personne qui a produit le code HTML ne connaît rien à PHP, et la personne qui programme en PHP n'a aucune idée de la mise en forme qui a été choisie. Intéressant non ?

Figure 6.4 — Affichage du document résultat

Avant d'instancier un template, chaque entité qui y est référencée doit se voir affecter une valeur. Comme le montre l'exemple ci-dessus, il existe trois façons de créer des entités et de leur affecter une valeur :

1. on charge un fichier avec `setFile()`, et on place son contenu dans une entité dont on fournit le nom ;
2. on effectue une simple affectation, comme dans `$vue->entite = valeur;` ;
3. on instancie un template, et on affecte le résultat à une entité ; pour cela on peut utiliser `assign()` qui remplace l'entité-cible, ou `append()` qui concatène la nouvelle valeur à celle déjà existant dans l'entité-cible.

6.3.2 Combiner des templates

Un moteur de templates serait bien faible s'il ne fournissait pas la possibilité de combiner des fragments pour créer des documents complexes. La combinaison des templates repose sur le mécanisme de remplacement d'entités. Il suffit de considérer que l'instanciation d'un template est une chaîne de caractères qui peut être constituer la valeur d'une nouvelle entité.

Prenons un autre exemple pour montrer la combinaison de templates. On veut produire un document affichant une liste dont on ne connaît pas à l'avance le nombre d'éléments. La figure 6.5 montre le résultat souhaité, avec 5 éléments dans la liste.

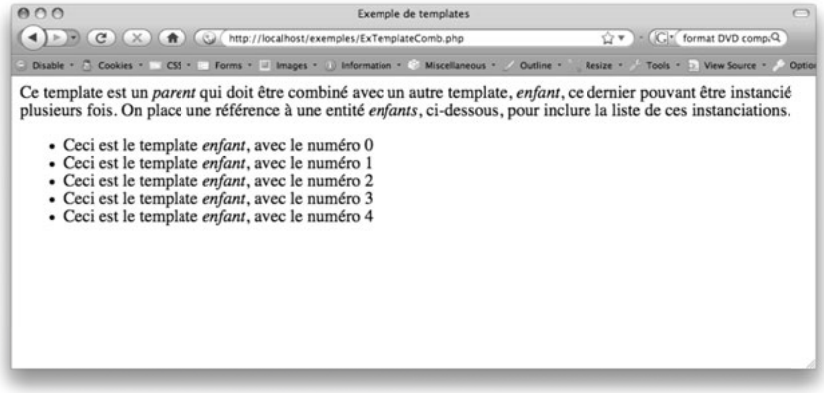

Exemple de templates
http://localhost/exemples/ExTemplateComb.php
Ce template est un parent qui doit être combiné avec un autre template, enfant, ce dernier pouvant être instancié plusieurs fois. On place une référence à une entité enfants, ci-dessous, pour inclure la liste de ces instanciations.
Ceci est le template enfant, avec le numéro 0
Ceci est le template enfant, avec le numéro 1
Ceci est le template enfant, avec le numéro 2
Ceci est le template enfant, avec le numéro 3
Ceci est le template enfant, avec le numéro 4

Figure 6.5 — Template contenant une liste

On ne peut pas obtenir ce résultat avec un seul template, parce qu'un des fragments (la première phrase) apparaît une seule fois, et l'autre partie (les éléments de la liste) plusieurs fois. La solution est de combiner deux templates. Voici le premier, le parent :

Exemple 6.5 *exemples/Parent.tpl* : *Template à instancier une fois*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Exemple de templates</title>

<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>
<div>
Ce template est un <i>parent</i>
 qui doit être combiné avec un autre template, <i>enfant</i>,
   ce dernier pouvant être instancié plusieurs fois.
   On place une référence à une entité <i>enfants</i>, ci-dessous
       ,
   pour inclure la liste de ces instanciations.
   <ul>
     {enfants}
    </ul>
</div>
</body>
</html>
```

Il contient la partie du document qui n'apparaît qu'une seule fois. La référence à l'entité `enfants` est destinée à être remplacée par la liste des éléments. Le second template représente un seul de ces éléments : on va ensuite concaténer les instanciations.

Exemple 6.6 *exemples/Enfant.tpl* : *Template à instancier autant de fois que nécessaire*

```
<li>
 Ceci est le template <i>enfant</i>, avec le numéro {numero}
</li>
```

Maintenant on peut effectuer la combinaison. Pour l'essentiel, on instancie autant de fois que nécessaire le template `enfant`, et on concatène ces instanciations dans une entité `enfants`. Au moment où on applique la méthode `render()`, la valeur d'`enfants` va se substituer à la référence vers cette entité dans `parent`, et le tour est joué.

Exemple 6.7 *exemples/ExTemplateComb.php* : *Le code PHP pour combiner les templates*

```
<?php
// Exemple de combinaison de templates
require ("Template.php");

// Instanciation d'un objet de la classe Template
$vue = new Template (".");

// Chargement des deux templates
$vue->setFile ("parent", "Parent.tpl");
```

```
$vue->setFile ("enfant", "Enfant.tpl");

// Boucle pour instancier 5 enfants
for ($i=0; $i < 5; $i++) {
  $vue->numero = $i;
  // On concatène l'instanciation de 'enfant' dans 'enfants'
  $vue->append ("enfants", "enfant");
 }
 // Et on affiche le résultat
 echo $vue->render("parent");
?>
```

Le mécanisme illustré ci-dessus peut sembler relativement complexe à première vue. Avec un peu de réflexion et d'usage, on comprend que les entités se manipulent comme des variables (chaînes de caractères). On les initialise, on les concatène et on les affiche. Cette approche permet de modifier à volonté la disposition de la page, sans qu'il soit nécessaire de toucher au code PHP, et inversement.

Un défaut potentiel des templates est qu'il faut parfois en utiliser beaucoup pour construire un document final complexe. Si on place chaque template dans un fichier dédié, on obtient beaucoup de fichiers, ce qui n'est jamais très facile à gérer. L'exemple ci-dessus est peu économe en nombre de fichiers puisque le template `enfant` tient sur 3 lignes.

Le mécanisme de *blocs* permet de placer plusieurs templates dans un même fichier. Le moteur de template offre une méthode, `setBlock()`, pour extraire un template d'un autre template, et le remplacer par une référence à une nouvelle entité. Avec `setBlock()`, on se ramène tout simplement à la situation où les templates sont dans des fichiers séparés.

Voici une illustration avec le même exemple que précédemment. Cette fois il n'y a plus qu'un seul fichier, avec deux templates :

Exemple 6.8 *exemples/ParentEnfant.tpl* : *Un fichier avec deux templates imbriqués*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Exemple de templates</title>

<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>
<div>
<div>
Ce template est un <i>parent</i>
 qui doit être combiné avec un autre template, <i>enfant</i>,
```

```
 directement inséré dans le même fichier.
   <ul>
     <!-- BEGIN enfant -->
    <li>
      Ceci est le template <i>enfant</i>, avec le numéro {numero}
    </li>
    <!--  END enfant -->
  </ul>
</div>
</body>
</html>
```

Le bloc correspondant au template `enfant` est imbriqué dans le premier avec une paire de commentaires HTML, et une syntaxe `BEGIN` – `END` marquant les limites du bloc. Voici maintenant le code PHP qui produit *exactement* le même résultat que le précédent.

Exemple 6.9 *exemples/ExTemplateBloc.php* : *Traitement d'un template avec bloc*

```
<?php
// Exemple de combinaison de templates avec bloc
require ("Template.php");

// Instanciation d'un objet de la classe Template
$vue = new Template (".");

// Chargement des deux templates
$vue->setFile ("parent", "ParentEnfant.tpl");

// On extrait le template 'enfant', et on le
// remplace par la référence à l'entité 'enfants'
$vue->setBlock ("parent", "enfant", "enfants");

// Boucle pour instancier 5 enfants
for ($i=0; $i < 5; $i++) {
  $vue->numero = $i;
  // On concatène l'instanciation de 'enfant' dans 'enfants'
  $vue->append ("enfants", "enfant");
 }
 // Et on affiche le résultat
 echo $vue->render("parent");
?>
```

Il faut noter qu'après l'appel à `setBlock()`, on se retrouve dans la même situation qu'après les deux appels à `setFile()` dans la version précédente. Ce que l'on a gagné, c'est l'économie d'un fichier.

6.3.3 Utilisation d'un moteur de templates comme vue MVC

Un moteur de templates est un bon candidat pour le composant « vue » d'une architecture MVC. Nous utilisons ce système de templates dans notre projet. Le chapitre 9 montrera une autre solution avec le *Zend Framework*. L'important, dans tous les cas, est de respecter le rôle de la vue, clairement séparée des actions et du modèle.

Dans notre MVC, chaque contrôleur dispose, par héritage, d'un objet `$this->vue`, instance de la classe `Template`. Cet objet charge les fichiers de templates à partir du répertoire *application/vues*. De plus, une entité nommée `page` est préchargée avec le document HTML de présentation du site. Ce document est beaucoup trop long pour être imprimé ici (vous pouvez bien sûr le consulter dans le code du site). Il nous suffit de savoir qu'il contient deux références à des entités `titre_page` et `contenu`. Chaque action doit donc construire un contenu pour ces entités et les affecter à la vue. À titre d'exemple, voici le contrôleur `index`, qui contient une seule action, `index`, affichant la page d'accueil.

Exemple 6.10 *webscope/application/controleurs/IndexCtrl.php* : *Le contrôleur* `index`

```
<?php
/**
 * @category webscope
 * @copyright Philippe Rigaux, 2008
 * @license GPL
 * @package Index
 */

require_once ("Controleur.php");

/**
 * Contrôleur par défaut: Index
 */

class IndexCtrl extends Controleur
{

  /**
   * Action par défaut
   */
  function index ()
  {
    /* Définition du titre */
    $this->vue->titre_page = "Accueil";

    /* On charge le code HTML de la page d'accueil
     * dans l'entité "contenu"
     */
    $this->vue->setFile("contenu", "index_accueil.tpl");

    /* Il n'y a plus qu'à afficher. NB: l'entité 'page' est
     définie dans la super-classe "Controleur" */
```

```
        echo $this->vue->render("page");
    }
}
?>
```

L'action se limite à définir les deux entités : `titre_page` est créé par une simple affectation, et `contenu` est créé par chargement du fichier template *index_accueil.tpl* qui contient le texte de la page d'accueil (pour mieux se repérer, les vues seront nommées d'après le contrôleur et l'action où elles sont utilisées). Il reste à appeler `render()` pour effectuer la substitution et obtenir l'affichage de la page d'accueil.

Cette solution garantit la séparation de PHP et HTML, puisqu'il est impossible de mettre du code PHP dans un template. Bien entendu, les choses vont se compliquer quand on va considérer des pages plus riches dans lesquelles les parties dynamiques produites par PHP vont elles-mêmes comporter une mise en forme HTML. L'exemple qui suit, plus réaliste, nous donnera une idée de la manière de metre en œuvre l'assocation entre les contrôleur/actions et la vue pour une fonctionnalité réelle.

6.3.4 Exemple complet

Nous allons créer, avec des *templates*, une fonctionnalité qui permet de rechercher des films pour les modifier. À partir de maintenant nous nous plaçons dans le cadre de la réalisation du site WEBSCOPE et nous concevons toute l'application comme un hiérarchie de contrôleurs et d'actions. Vous pouvez ,en parallèle de votre lecture, consulter ou modifier le code fourni sur notre site ou sur le serveur de SourceForge.

Le contrôleur s'appelle `saisie` et la fonctionnalité de recherche est composée de deux actions : `form_recherche` et `recherche`. Vous savez maintenant où trouver le code correspondant : le contrôleur est une classe `SaisieCtrl.php` dans *application/controleurs*, et les deux actions correspondent à deux méthodes de même nom.

La première action se déclenche avec l'URL

index.php?ctrl=saisie&action=form_recherche

ou plus simplement *?ctrl=saisie&action=form_recherche* quand on est déjà dans le contexte de l'application *webscope*. Elle affiche un formulaire pour saisir un mot-clé, complet ou partiel, correspondant à une sous-chaîne du titre des films recherchés (voir la figure 6.6).

La seconde action (figure 6.7) montre un tableau contenant, après recherche, les films trouvés, associés à une ancre permettant d'accéder au formulaire de mise à jour (non décrit ici). Dans notre copie d'écran, on a demandé par exemple tous les films dont le titre contient la lettre « w » pour trouver *Sleepy Hollow*, *Eyes Wide Shut*, *King of New York*, etc.

Pour chaque action nous disposons d'un template. D'une manière générale, c'est une bonne habitude d'essayer de conserver un template par action et de nommer les fichiers de templates d'après l'action et le contrôleur. Dans notre cas les fichiers s'appellent respectivement *saisie_form_recherche.tpl* et *saisie_recherche.tpl*.

Figure 6.6 — Page de recherche des films

Figure 6.7 — Le résultat d'une recherche

Voici le premier :

Exemple 6.11 *Le template saisie_form_recherche.tpl affichant le formulaire de recherche*

```
<p>
Vous pouvez rechercher avec ce formulaire les films
que vous souhaitez modifier. Entrez le titre, ou
une partie du titre, en majuscules ou minuscules,
et lancez la recherche.
</p>
```

```
<!— Le formulaire pour saisir la requête —>

<center>
<form  method='post' action='?ctrl=saisie&amp;action=recherche'>
 <b>Titre ou partie du titre</b>
  <input type='text' name="titre" value=""  size='30' maxlength
      ='30'/>
  <input type='submit' name="submit" value="Rechercher"/>
</form>
</center>
```

Rappelons que notre « *layout* » comprend deux références d'entités : `titre_page` et `contenu` (voir ce qui précède). Le but de chaque action (au moins en ce qui concerne la présentation du résultat) est de créer une valeur pour ces deux entités. Voici l'action `form_recherche`.

```
function form_recherche ()
 {
   /* Définition du titre */
   $this->vue->titre_page = "Recherche des films";

   /**
    * On charge le template "saisie_recherche"
    * dans l'entité "contenu"
    */
   $this->vue->setFile("contenu", "saisie_form_recherche.tpl");

   /* Il n'y a plus qu'à afficher. */
   echo $this->vue->render("page");
 }
```

C'est une page statique, qui se contente de combiner deux templates en plaçant le contenu du fichier *saisie_form_recherche.tpl* dans l'entité `contenu` du *layout*. La seconde action est un peu plus longue (forcément). Voyons d'abord le template :

Exemple 6.12 *Le template saisie_recherche.tpl montrant le résultat de la recherche*

```
<p>
<b>Voici le résultat de votre recherche. </b> Vous
pouvez maintenant utiliser le lien "mise à jour"
pour accéder à un formulaire de modification des films.
</p>
<center>
  <table border='4' cellspacing='5'>
  <tr class="header">
   <th>Titre</th><th>Année</th><th>Action</th>
  </tr>
  <!—  Le bloc pour le template affichant une ligne —>
  <!— BEGIN ligne —>
```

```
  <tr class='{classe_css}'>
    <td>{titre_film}</td><td>{annee}</td>
    <td><a href="?ctrl=saisie&amp;action=form_modifier&amp;id={
        id_film}">
            Mise à jour</a>
    </td>
  </tr>
  <!-- END ligne -->
  </table>
</center>
```

Il s'agit de deux templates imbriqués. Le second, marqué par les commentaires `BEGIN` et `END`, correspond à chaque ligne du tableau montrant les films. À l'intérieur de ce template imbriqué on trouve les références aux entités `classe_css`, `titre_film`, `id_film`, et `annee`. Le code de l'action est donné ci-dessous : les commentaires indiquent le rôle de chaque partie.

```
function recherche ()
 {
   // Définition du titre
   $this->vue->titre_page = "Résultat de la recherche";

   // On charge les templates nécessaires
   $this->vue->setFile("texte", "saisie_recherche.tpl");

   // On extrait le bloc imbriqué "ligne", et on le remplace par
   // la référence à une entité "lignes"
   $this->vue->setBlock ("texte", "ligne", "lignes");

   // Le titre a été saisi ? On effectue la recherche
   if (isSet($_POST['titre'])) {
    $titre = htmlEntities($_POST['titre']);
    }
   else {
     // Il faudrait sans doute protester?
     $titre = "";
   }

   // Exécution de la requête
   $requete = "SELECT * FROM Film WHERE titre LIKE '%$titre%' ";
   $resultat = $this->bd->execRequete($requete);

   $compteur = 1;
   while ($film = $this->bd->objetSuivant ($resultat)) {
     if ($compteur++ % 2 == 0) $classe = "even"; else $classe =
         "odd";

     // Affectation des entités de la ligne
     $this->vue->classe_css = $classe;
     $this->vue->titre_film = $film->titre;
     $this->vue->id_film = $film->id;
```

```
        $this->vue->annee = $film->annee;

        // On effectue la substitution dans "ligne", en concaténant
        // le résultat dans l'entité "lignes"
        $this->vue->append ("lignes", "ligne");
    }

    // On a le formulaire et le tableau: on parse et on place
    // le résultat dans l'entité 'contenu'
    $this->vue->assign("contenu", "texte");

    /* Il n'y a plus qu'à afficher. */
    echo $this->vue->render("page");
}
```

Notez que l'action attend en paramètre une variable `titre` transmise en `post`. En principe ce paramètre vient du formulaire. Une action devrait toujours vérifier que les paramètres attendus sont bien là, et filtrer leur valeur (en supprimant par exemple les balises HTML que des personnes malicieuses pourraient y injecter). C'est ce que fait la fonction `htmlEntities()`, en remplaçant les caractères réservés HTML par des appels d'entités. Rappelez-vous toujours qu'un script PHP peut être déclenché par n'importe qui, et pas toujours avec de bonnes intentions.

Ces actions sont du « pur » PHP, sans aucune trace de HTML. Si on conçoit les choses avec soin, on peut structurer ainsi une application MVC en fragments de code, chacun d'une taille raisonnable, avec une grande clarté dans l'organisation de toutes les parties de l'application. Avant d'étudier la dernière partie du MVC, le modèle, nous allons comme promis revenir un moment sur les avantages et inconvénients du système de templates pour gérer le composant Vue.

6.3.5 Discussion

Les *templates* offrent un bon exemple de la séparation complète de la « logique » de l'application, codée en PHP, et de la présentation, codée en HTML. Une des forces de ce genre de système est que toute la mise en forme HTML est écrite une seule fois, puis reprise et manipulée grâce aux fonctions PHP. On évite donc, pour les modifications du site, l'écueil qui consisterait à dupliquer une mise en forme autant de fois qu'il y a de pages dans le site. C'est ce que doit satisfaire tout gestionnaire de contenu HTML digne de ce nom en proposant une notion de « style » ou de « modèle » dont la mise à jour est répercutée sur toutes les pages reposant sur ce style ou ce modèle.

Un problème délicat reste la nécessité de produire un nombre très important de *templates* si on veut gérer la totalité du site de cette manière et interdire la production de tout code HTML avec PHP. Cette multiplication de « petits » modèles (pour les tableaux, les lignes de tableaux, les formulaires et tous leurs types de champs, etc.) peut finir par être très lourde à gérer. Imaginez par exemple ce que peut être la production avec des *templates* d'un formulaire comme ceux que nous pouvons obtenir avec la classe *Formulaire*, comprenant une imbrication de tableaux, de champs de saisie et de valeurs par défauts fournies en PHP.

Un bon compromis est d'utiliser des modèles de page créés avec un générateur de documents HTML, pour la description du graphisme du style. Cela correspond *grosso modo* à l'en-tête, au pied de page et aux tableaux HTML qui définissent l'emplacement des différentes parties d'une page. On place dans ces modèles des entités qui définissent les composants instanciés par le script PHP : tableaux, formulaires, menus dynamiques, etc. Ensuite, dans le cadre de la programmation PHP, on prend ces modèles comme *templates*, ce qui rend le code indépendant du graphisme, et on utilise, pour produire le reste des éléments HTML, plus neutres du point de vue de la présentation, les utilitaires produisant des objets HTML complexes comme `Tableau` ou `Formulaire`.

Le code PHP produit alors ponctuellement des composants de la page HTML, mais dans un cadre bien délimité et avec des utilitaires qui simplifient beaucoup cette tâche. L'utilisation des feuilles de style CSS permet de gérer quand même la présentation de ces éléments HTML. Il suffit pour cela de prévoir l'ajout d'une classe CSS dans les balises HTML produites. Cette solution limite le nombre de templates nécessaires, tout en préservant un code très lisible.

On peut également s'intéresser à des systèmes de *templates* plus évolués que celui présenté ici. Il en existe beaucoup (trop ...). Attention cependant : le choix d'un système de *templates* a un impact sur tout le code du site, et il n'est pas du tout facile de faire marche arrière si on s'aperçoit qu'on a fait fausse route. Posez-vous les quelques questions suivantes avant de faire un choix :

- Le système préserve-t-il la simplicité de production du code HTML, ou faut-il commencer à introduire des syntaxes compliquées dans les *templates* pour décrire des boucles, des éléments de formulaire, etc. La méthode consistant à décrire des blocs est un premier pas vers l'introduction de structures de programmation (boucles, tests) dans les modèles, et il est tentant d'aller au-delà. Si la personne responsable du code HTML doit se transformer en programmeur, on perd cependant l'idée de départ...
- Le système est-il répandu, bien documenté, soutenu par une collectivité active et nombreuse de programmeurs ? Est-il, au moins en partie, compatible avec les systèmes classiques ?
- Quelles sont ses performances ? Est-il doté d'un système de *cache* qui évite d'effectuer systématiquement les opérations coûteuses de substitution et de copies de chaînes de caractères ?

Gardez en vue qu'un bon système de *templates* doit avant tout faciliter la répartition des tâches et rester simple et efficace. Il paraît peu raisonnable de se lancer dans des solutions sans doute astucieuses mais complexes et non normalisées. Si vraiment la séparation du contenu et de la présentation est très importante pour vous, par exemple parce que vous souhaitez produire plusieurs formats différents (HTML, WML, PDF, etc.) à partir d'un même contenu, pourquoi ne pas étudier les outils basés sur XML comme le langage de transformation XSLT, introduit dans le chapitre 8 ? Ces langages sont normalisés par le W3C, on bénéficie donc en les adoptant des très nombreux outils et environnements de développement qui leur sont consacrés.

Nous verrons également dans le chapitre 9 une approche pour gérer la vue, celle du *Zend Framework*, assez différente des systèmes de templates. Elle a le mérite d'utiliser directement PHP pour la mise en forme, ce qui évite d'avoir à inventer un nouveau pseudo-langage de programmation. En contrepartie la saisie est lourde et le code obtenu peu plaisant à lire. Le système idéal, simple, léger, lisible et bien intégré à PHP, reste à définir.

En résumé, le style d'imbrication de PHP et de HTML fait partie des questions importantes à soulever avant le début du développement d'un site. La réponse varie en fonction de la taille du développement et de l'équipe chargée de la réalisation, des outils disponibles, des compétences de chacun, des contraintes (le site doit-il évoluer fréquemment ? Doit-il devenir multilingue à terme, certaines fonctionnalités sont-elles communes à d'autre sites ?), etc. J'espère que les différentes techniques présentées dans ce livre vous aideront à faire vos choix en connaissance de cause.

6.4 STRUCTURE D'UNE APPLICATION MVC : LE MODÈLE

Il nous reste à voir le troisième composant de l'architecture MVC : le modèle. Le modèle est constitué de l'ensemble des fonctionnalités relevant du traitement (au sens large) des données manipulées par l'application. Cette notion de traitement exclut la présentation qui, nous l'avons vu, est prise en charge par la vue. Tout ce qui concerne la gestion des interactions avec l'utilisateur ainsi que le *workflow* (séquence des opérations) relève du contrôleur. Par élimination, tout le reste peut être imputé au modèle. Il faut souligner qu'on y gagne de ne pas du tout se soucier, en réalisant le modèle, du contexte dans lequel il sera utilisé. Un modèle bien conçu et implanté peut être intégré à une application web mais doit pouvoir être réutilisé dans une application client/serveur, ou un traitement *batch*. On peut le réaliser de manière standard, sous forme de fonctions ou de classes orientées-objet, sans se soucier de HTML. Il n'y aurait pas grand-chose de plus à en dire si, très souvent, le modèle n'était pas également le composant chargé d'assurer la *persistance* des données, autrement dit leur survie indépendamment du fonctionnement de l'application.

6.4.1 Modèle et base de données : la classe `TableBD`

Dans des applications web dynamiques, le modèle est aussi une couche d'échange entre l'application et la base de données. Cette couche peut simplement consister en requêtes SQL de recherche et de mise à jour. Elle peut être un peu plus sophistiquée et factoriser les fonctions assurant les tâches routinières : recherche par clé, insertion, mise à jour, etc. À l'extrême, on peut mettre en œuvre un *mapping* objet-relationnel (*Objet-Relational Mapping*, ORM en anglais) qui propose une vue de la base de données reposant sur des classes orientées-objet. Ces classes masquent le système relationnel sous-jacent, ainsi que les requêtes SQL.

Comme d'habitude, essayons d'être simples et concret : dans ce qui suit je propose une couche Modèle un peu plus élaborée que la communication par SQL, et je montre comment l'exploiter dans notre site pour des recherches (pas trop sophistiquées) et des mises à jour. Le chapitre 9 montre avec le Zend Framework le degré d'abstraction que l'on peut obtenir avec une couche ORM.

Nous allons reprendre la classe générique `IhmBD` déjà présentée partiellement dans le chapitre 3, consacré à la programmation objet (voir page 167) et l'étendre dans l'optique d'un Modèle MVC aux aspects propres à la recherche et à la mise à jour de la base. Elle s'appellera maintenant `TableBD`. Le tableau 6.2 donne la liste des méthodes génériques assurant ces fonctions (ce tableau est complémentaire de celui déjà donné page 170).

Tableau 6.2 — Les méthodes d'interaction avec la base de la classe `TableBD`

Méthode	Description
nouveau(*données*)	Création d'un nouvel objet.
chercherParCle(*clé*)	Recherche d'une ligne par clé primaire.
controle()	Contrôle les valeurs avant mise à jour.
insertion(*données*)	Insertion d'une ligne.
maj(*données*)	Mise à jour d'une ligne.

Conversion des données de la base vers une instance de `TableBD`

La classe (ou plus exatement les objets instance de la classe) vont nous servir à interagir avec une table particulière de la base de données. Le but est de pouvoir manipuler les données grâce aux méthodes de la classe, en recherche comme en insertion. La première étape consiste à récupérer le schéma de la table pour connaître la liste des attributs et leurs propriétés (type, ou contraintes de clés et autres). Il faut aussi être en mesure de stocker une ligne de la table, avant une insertion ou après une recherche. Pour cela nous utilisons deux tableaux, pour le schéma, et pour les données.

```
protected $schema = array (); // Le schéma de la table
protected $donnees = array(); // Les données d'une ligne
```

La déclaration `protected` assure que ces tableaux ne sont pas accessibles par une application interagissant avec une instance de la classe. En revanche ils peuvent être modifiés ou redéfinis par des instances d'une sous-classe de `TableBD`. Comme nous le verrons, `TableBD` fournit des méthodes génériques qui peuvent être spécialisées par des sous-classes.

Pour obtenir le schéma d'une table nous avons deux solutions : soit l'indiquer explicitement, en PHP, pour chaque table, soit le récupérer automatiquement en interrogeant le serveur de données. Notre classe `BD` dispose déjà d'une méthode, `schemaTable()`, qui récupère le schéma d'une table sous forme de tableau (voir page 132). Nous allons l'utiliser. Cette méthode prend en entrée un nom de table et retourne un tableau comportant une entrée par attribut. Voici par exemple ce que l'on obtient pour la table *Internaute*.

```
Array
(
    [email] => Array (
       [longueur] => 40  [type] => string [clePrimaire] => 1  [
           notNull] => 1
        )
```

```
    [nom] => Array (
       [longueur] => 30 [type] => string [clePrimaire] => 0 [
          notNull] => 1
        )

    [prenom] => Array (
       [longueur] => 30 [type] => string [clePrimaire] => 0[
          notNull] => 1
        )

    [mot_de_passe] => Array (
       [longueur] => 32] [type] => string [clePrimaire] => 0 [
          notNull] => 1
        )

    [annee_naissance] => Array (
       [longueur] => 11 [type] => int [clePrimaire] => 0 [notNull
          ] => 0
        )

    [region] => Array (
       [longueur] => 30 [type] => string [clePrimaire] => 0 [
          notNull] => 0
        )
)
```

Ces informations nous suffiront pour construire la classe générique. Notez en particulier que l'on connaît le ou les attributs qui constituent la clé primaire (ici, l'e-mail). Cette information est indispensable pour chercher des données par clé ou effectuer des mises à jour. Le constructeur de `TableBD` recherche donc le schéma de la table-cible et initialise le tableau `donnees` avec des chaînes vides.

```
 function __construct ($nom_table, $bd, $script="moi")
{
  // Initialisation des variables privées
  $this->bd = $bd;
  $this->nom_table = $nom_table;

  // Lecture du schéma de la table, et lancer d'exception si
     problème
  $this->schema = $bd->schemaTable($nom_table);

  // On initialise le tableau des données
  foreach ($this->schema as $nom => $options) {
    $this->donnees[$nom] = "";
   }
}
```

Le tableau des données est un simple tableau associatif, dont les clés sont les noms des attributs, et les valeurs de ceux-ci. Après l'appel au constructeur, ce tableau des données est vide. On peut l'initialiser avec la méthode `nouveau()`, qui prend en

entrée un tableau de valeurs. On extrait de ce tableau les valeurs des attributs connus dans le schéma, et on ignore les autres. Comme l'indiquent les commentaires dans le code ci-dessous, il manque de nombreux contrôles, mais l'essentiel de la fonction d'initialisation est là.

```
/**
 * Méthode créant un nouvel objet à partir d'un tableau
 */

public function nouveau ($ligne)
{
  // On parcourt le schéma. Si, pour un attribut donné,
  // une valeur existe dans le tableau: on l'affecte

  foreach ($this->schema as $nom => $options)
  {
    // Il manque beaucoup de contrôles. Et si $ligne[$nom]
    // était un tableau ?
    if (isSet($ligne[$nom]))
      $this->donnees[$nom] = $ligne[$nom];
  }
}
```

Une autre manière d'initialiser le tableau des données est de rechercher dans la base, en fonction d'une valeur de clé primaire. C'est ce que fait la méthode `chercherParCle()`.

```
public function chercherParCle ($cle)
{
  // Commençons par chercher la ligne dans la table
  $clauseWhere = $this->accesCle ($params, "SQL");

  // Création et exécution de la requête SQL
  $requete = "SELECT * FROM $this->nom_table WHERE $clauseWhere
      ";
  $resultat = $this->bd->execRequete($requete);
  $ligne = $this->bd->ligneSuivante($resultat);

  // Si on n'a pas trouvé, c'est que la clé n'existe pas:
  // on lève une exception
  if (!is_array($ligne)) {
    throw new Exception ("TableBD::chercherParCle. La ligne n'
        existe pas.");
  }
  // Il ne reste plus qu'à créer l'objet avec les données du
      tableau
  $this->nouveau($ligne);

  return $ligne;
}
```

La méthode reçoit les valeurs de la clé dans un tableau, constitue une clause `WHERE` (avec une méthode `accesCle()` que je vous laisse consulter dans le code

lui-même), et initialise enfin le tableau des données avec la méthode `nouveau()`, vue précédemment. Une exception est levée si la clé n'est pas trouvée dans la base.

Finalement, comment accéder individuellement aux valeurs des attributs pour une ligne donnée ? On pourrait renvoyer le tableau `donnees`, mais ce ne serait pas très pratique à manipuler, et romprait le principe d'encapsulation qui préconise de ne pas dévoiler la structure interne d'un objet.

On pourrait créer des *accesseurs* nommés `get`*`Nom`*`()`, où *`nom`* est le nom de l'attribut dont on veut récupérer la valeur. C'est propre, mais la création un par un de ces accesseurs est fastidieuse.

PHP fournit un moyen très pratique de résoudre le problème avec des méthodes dites *magiques*. Elles permettent de coder une seule fois les accesseurs `get` et `set`, et prennent simplement en entrée le nom de l'attribut visé. Voici ces méthodes pour notre classe générique.

```
/**
 * Méthode "magique" get: renvoie un élément du tableau
 * de données
 */

public function __get ($nom)
{
  // On vérifie que le nom est bien un nom d'attribut du schéma
  if (! in_array($nom, array_keys($this->schema))) {
    throw new Exception ("$nom n'est pas un attribut de la
        table $this->nom_table");
  }

  // On renvoie la valeur du tableau
  return $this->donnees[$nom];
}

/**
 * Méthode "magique" set: affecte une valeur à un élément
 * du tableau de données
 */

public function __set ($nom, $valeur)
{
  // On vérifie que le nom est bien un nom d'attribut du schéma
  if (! in_array($nom, array_keys($this->schema))) {
    throw new Exception ("$nom n'est pas un attribut de la
        table $this->nom_table");
  }

  // On affecte la valeur au tableau (des contrôles seraient
  // bienvenus...)
  $this->donnees[$nom] = $valeur;
}
```

La méthode `__get(nom)` est appelée chaque fois que l'on utilise la syntaxe `$o->nom` pour *lire* une propriété qui n'existe pas explicitement dans la classe. Dans notre cas, cette méthode va simplement chercher l'entrée *nom* dans le tableau de données. La méthode `__set(nom, valeur)` est appelée quand on utilise la même syntaxe pour réaliser une affectation. Ces méthodes magiques masquent la structure interne (que l'on peut donc modifier de manière transparente) en évitant de reproduire le même code pour tous les accesseurs nécessaires. Il existe également une méthode magique `__call(nom, params)` qui intercepte tous les appels à une méthode qui n'existe pas.

Contrôles, insertion et mises à jour

Maintenant que nous savons comment manipuler les valeurs d'un objet associé à une ligne de la table, il reste à effectuer les contrôles et les mises à jour. La méthode `controle()` vérifie les types de données et la longueur des données à insérer. La contrainte de généricité interdit d'aller bien plus loin.

```
protected function controle()
 {
   // On commence par vérifier les types de données
   foreach ($this->schema as $nom => $options) {
     // Contrôle selon le type de l'attribut
     if ($options['type'] == "string") {
       // C'est une chaîne de caractères. Vérifions sa taille
       if (strlen ($this->donnees[$nom]) > $options['longueur'])
           {
         $this->erreurs[] = "La valeur pour $nom est trop longue
             ";
         return false;
       }
     }
     else if ($options['type'] == "int") {
       // Il faut que ce soit un entier
       if (!is_int($this->donnees[$nom])) {
         $this->erreurs[] = "$nom doit être un entier";
         return false;
       }
     }
     return true;
   }
 }
```

Les méthodes d'insertion et de mise à jour fonctionnent toutes deux sur le même principe. On construit dynamiquement la requête SQL (`INSERT` ou `UPDATE`), puis on l'exécute. L'exemple de l'insertion est donné ci-dessous. Bien entendu on exploite le schéma de la table pour connaître le nom des attributs, et on trouve les valeurs dans le tableau de données.

```
public function insertion ()
{
  // Initialisations
  $noms = $valeurs = $virgule = "";
```

```
    // Parcours des attributs pour créer la requête
    foreach ($this->schema as $nom => $options) {
      // Liste des noms d'attributs + liste des valeurs (
          attention aux ')
      $noms .= $virgule . $nom;
      $valeur = $this->bd->prepareChaine($this->donnees[$nom]) ;
      $valeurs .= $virgule . "'$valeur'";
      // A partir de la seconde fois, on sépare par des virgules
      $virgule= ",";
    }
    $requete = "INSERT INTO $this->nom_table($noms) VALUES (
        $valeurs) ";
    $this->bd->execRequete ($requete);
  }
```

La fonction de mise à jour est similaire ; je vous laisse la consulter dans le code lui-même. Nous voici équipés avec un cadre pré-établi pour réaliser la partie Modèle d'une application. Outre l'intérêt de disposer de fonctionnalités prêtes à l'emploi, ce qui peut déjà économiser du développement, cette approche a aussi le mérite de normaliser les méthodes de programmation, avec gain de temps là encore quand on consulte le code.

6.4.2 Un exemple complet de saisie et validation de données

Montrons pour conclure ce chapitre comment réaliser une fonctionnalité complète MVC, incluant une partie Modèle pour communiquer avec la base de données. L'exemple choisi est celui de l'inscription d'un internaute sur le site. On demande de saisir ses données personnelles dans un formulaire, y compris un mot de passe d'accès au site, pour lequel on demande une double saisie. La validation de ce formulaire entraîne une insertion dans la base. La figure 6.8 montre le formulaire d'inscription.

Figure 6.8 — Formulaire d'inscription sur le site

Le contrôleur en charge des fonctionnalités d'inscription est `inscription`. L'action par défaut (`index`) affiche le formulaire de la figure 6.8. Voici son code.

```
function index ()
 {
   // On affecte le titre et on charge le contenu
   $this->vue->titre_page = "Inscription";
   $this->vue->setFile("contenu", "inscription.tpl");

   // On instancie la classe TableBD sur 'Internaute'
   $tbl_inter = new Internaute ($this->bd);

   // Production du formulaire en insertion
   $this->vue->formulaire = $tbl_inter->formulaire(TableBD::
       INS_BD,
                                  "inscription", "enregistrer");

   echo $this->vue->render("page");
 }
```

On instancie un objet `$tbl_inter` de la classe `Internaute` (le fichier *Internaute.php* se trouve dans *application/modeles*). Cette classe est une sous-classe de `TableBD`, hérite de toutes ses fonctionnalités et en redéfinit quelques-unes pour s'adapter aux spécificités de manipulation des données de la table *Internaute*.

La première particularité est le constructeur. Comme on sait sur quelle table s'appuie la classe, on peut passer son nom « en dur » au constructeur de `TableBD`, ce qui donne le code ci-dessous.

```
class Internaute extends TableBD
{
  // Le constructeur de la classe. On appelle
  // simplement le constructeur de la super-classe.
  // en lui passant le nom de la table visée.

 function __construct($bd)
  {
    // Appel du constructeur de IhmBD
    parent::__construct("Internaute", $bd);
   }
```

La seconde particularité est le formulaire de saisie. On ne peut pas se contenter du formulaire générique proposé par `TableBD` car il faut demander deux fois le mot de passe à l'utilisateur afin d'avoir confirmation qu'il n'a pas commis d'erreur de saisie. Il faut donc redéfinir dans `Internaute` la méthode `Formulaire()` qui vient remplacer (« surcharger » est le terme juste) celle héritée de `TableBD`. Nous avons déjà vu à plusieurs reprises comment produire des formulaires de saisie, je vous laisse consulter cette méthode dans le code.

Rappelons que dans une application de base de données, une grande partie des formulaires est destinée à effectuer des opérations *d'insertion* ou de *mise à jour* sur les tables. Bien entendu, il faut éviter d'utiliser un formulaire distinct pour chacune

de ces opérations et nous utilisons donc la technique détaillée dans le chapitre 2, page 78, pour adapter la présentation des champs en fonction du type d'opération effectué.

Passons à la mise à jour. Ici encore les fonctions génériques fournies par `TableBD` ne suffisent pas. C'est notamment le cas de la méthode `controle()` qui comprend de nombreux contrôles complémentaires de ceux effectués dans la méthode générique. La complémentarité implique que la méthode de la super-classe doit être appelée *en plus* des contrôles ajoutés dans la méthode spécialisée. Cela se fait en plaçant explicitement un appel `parent::controle` dans le code, comme montré ci-dessous :

```
function controle ()
{
  // Initialisation de la liste des messages d'erreur
  $this->erreurs = array();

  // On vérifie que les champs importants ont été saisis
  if ($this->donnees['email']=="")
    $this->erreurs[] = "Vous devez saisir votre e-mail !";
  else if (!$this->controleEmail($this->donnees['email']))
    $this->erreurs[]  = "Votre e-mail doit être de la forme
        xxx@yyy[.zzz] !";

  // Contrôle sur le mot de passe
  if (isSet ($this->donnees['mot_de_passe'])) {
    if ($this->donnees['mot_de_passe']==""
    or $_POST['conf_passe']==""
    or $_POST['conf_passe'] != $this->donnees['mot_de_passe'])
      $this->erreurs[]  .= "Vous devez saisir un mot de passe "
       . " et le confirmer à l'identique!";
  }

  if (!isSet($this->donnees['region']) or empty($this->donnees[
      'region']))
    $this->erreurs[]  .= "Vous devez saisir votre région !";
  if ($this->donnees['annee_naissance']=="")
    $this->erreurs[]  .= "Votre année de naisance est
        incorrecte!";
  if ($this->donnees['prenom']=="")
    $this->erreurs[]  .= "Vous devez saisir votre prénom !";
  if ($this->donnees['nom']=="")
    $this->erreurs[]  .= "Vous devez saisir votre nom !";

  // Appel aux contrôles de la méthode générique
  parent::controle();

  if (count($this->erreurs) > 0) {
    return false;
  }
  else {
    return true;
  }
}
```

On peut toujours ajouter ou raffiner des contrôles. Vous pouvez vous reporter à la section consacrée à la validation des formulaires, page 86, pour un exposé des différentes vérifications nécessaires.

Une autre méthode modifiée par rapport à la méthode générique est `insertion()`. Le seul ajout est le hachage du mot de passe avec la fonction MD5, afin de ne pas l'insérer en clair dans la base.

```
function insertion()
{
  // On insère le mot de passe haché
  $this->donnees['mot_de_passe'] = md5 ($this->donnees
     ['mot_de_passe']);

  // Il ne reste plus qu'à appeler la méthode d'insertion
      héritée
  parent::insertion();
}
```

Pour finir, voici l'action `enregistrer` du contrôleur `Inscription`. C'est cette action qui est appelée quand on valide le formulaire.

```
 function enregistrer ()
{
  // Idem que précédemment
  $this->vue->titre_page = "Résultat de votre inscription";
  $tbl_inter = new Internaute ($this->bd);

  // On crée un objet à partir des données du tableau HTTP
  $tbl_inter->nouveau($_POST);

  // Contrôle des variables passées en POST
  if ( $tbl_inter->controle() == false) {
    $messages = $tbl_inter->messages();
    // Erreur de saisie détectée: on affiche le message
    // et on réaffiche le formulaire avec les valeurs saisies
    $this->vue->contenu =  "<b>$messages</b>\n"
    . $tbl_inter->formulaire (TableBD::INS_BD,
           "inscription", "enregistrer");
  }
  else {
    // On va quand même vérifier que cet email n'est pas déjà
    // inséré
    if ($inter = $tbl_inter->chercheLigne($_POST)) {
      $this->vue->contenu = "Un internaute avec cet email
          existe déjà "
      . $tbl_inter->formulaire (TableBD::INS_BD, "inscription",
          "enregistrer");
    }
    else {
      $tbl_inter->insertion();
```

```
            // Message de confirmation
            $this->vue->contenu = "Vous êtes bien enregistré avec
                l'email "
            . "<b>$tbl_inter->email</b>. Bienvenue!<br/>"
            . "Vous pouvez maintenant vous connecter au site.";
        }
    }

    // Finalement, on affiche la vue comme d'habitude
    echo $this->vue->render("page");
}
```

Après initialisation d'une instance de la classe `Internaute`, l'action débute par l'exécution de la méthode `controle()`. Si une erreur est détectée, un message est constitué et on réaffiche le formulaire en reprenant, pour valeurs par défaut, les saisies présentes dans le tableau `$_POST`. Sinon, on vérifie que l'e-mail n'existe pas déjà, puis on insère.

6.4.3 Pour conclure

Ce dernier exemple a montré de manière complète l'interaction des trois composants du MVC. La structuration en contrôleurs et actions permet de situer facilement le déroulement d'une suite d'actions (ici, saisie, mise à jour) et fournit une initialisation de l'environnement (comme la connexion à la base) qui épargne au programmeur les instructions répétitives. La vue est en charge de la sortie HTML, et on constate que les actions ne contiennent plus aucune balise HTML. Le modèle, au moins dans la prise en compte de la persistance, fournit encore des fonctionnalités toutes prêtes qui limitent la taille du code et facilitent sa compréhension.

Convaincu(e) ? Comme tout concept un peu avancé, le MVC demande un peu de pratique et un délai d'assimilation. Cet effort en vaut vraiment la peine, et ce chapitre avait pour but de vous proposer une introduction la plus douce possible, tout en montrant une implantation réaliste. Prenez le temps d'analyser soigneusement la fonctionnalité d'inscription pour bien comprendre les interactions entre les différents composants. Le découpage induit par le MVC est logique, cohérent, et mène à des fragments de code tout à fait maîtrisables par leur taille et leur complexité limitée.

Le reste du site est constitué d'actions qui peuvent s'étudier isolément, indépendamment les unes des autres et indépendamment du contexte MVC. Encore une fois, récupérez le code du site, étudiez-le et modifiez-le. Quand vous aurez assimilé les principes, vous pourrez passer à des fonctionnalités plus poussées et à des *frameworks* de développement plus robustes.

En ce qui concerne la complexité du développement MVC, il faut prendre conscience que les objets `Internaute` manipulés pour l'inscription sont très simples et correspondent à la situation élémentaire où la correspondance établie par le modèle associe un objet (instance de la classe `Internaute`) à une seule ligne d'une table (*Internaute*). C'est un cas de *mapping* (correspondance entre deux représentations) trivial. Vous trouverez dans le code du site une version plus complexe d'un

modèle représentant les films. Un film est modélisé comme un objet composé de lignes provenant de plusieurs tables : les données du film lui-même (table *Film*), le metteur en scène (table *Artiste*) et la liste des acteurs (table *Artiste* également). Tout en gardant la même interface simple que `TableBD`, la classe `Film` gère ce *mapping* en reconstituant la description complète d'un film comme un graphe d'objets au moment des recherches ou des mises à jour. Les nombreux commentaires placés dans le code vous permettront de comprendre l'articulation des données.

Enfin, je vous rappelle que le chapitre 9 est consacré à une introduction au *Zend Framework* qui constitue une réalisation d'une toute autre envergure et d'une toute autre complexité que le MVC simplifié présenté ici.

7

Production du site

Ce chapitre est consacré aux fonctionnalités du site WEBSCOPE. Elles sont développées selon les principes décrits dans les chapitres précédents. Le site s'appuie sur la base de données définie au chapitre 4. Rappelons que vous pouvez, d'une part utiliser ce site sur notre serveur, d'autre part récupérer la totalité du code, le tester ou le modifier à votre convenance. Rappelons enfin que ce code est conçu comme une application PHP fonctionnant aussi bien avec MySQL que n'importe quel SGBD relationnel. Le chapitre aborde successivement quatre aspects, correspondant chacun à un contrôleur.

Le premier (contrôleur `Auth`) est la gestion de *l'authentification* permettant d'identifier un internaute accédant au site, avant de lui accorder des droits de consultation ou de mise à jour. Ces droits sont accordés pour une *session* d'une durée limitée, comme présenté déjà page 98. Cette fonctionnalité d'authentification couplée avec une gestion de sessions est commune à la plupart des sites web interactifs.

Les points suivants sont plus spécifiques, au moins du point de vue applicatif, au site de filtrage coopératif WEBSCOPE. Nous décrivons tout d'abord le contrôleur `Notation` qui permet de rechercher des films et de leur attribuer une note, puis l'affichage des films (contrôleur `Film`) avec toutes leurs informations. Cet affichage comprend un forum de discussion qui permet de déposer des commentaires sur les films et de répondre aux commentaires d'autres internautes.

Enfin, la dernière partie (contrôleur `Recomm`) est consacrée à l'algorithme de prédiction qui, étant données les notes déjà attribuées par un internaute, recherche les films les plus susceptibles de lui plaire (contrôleur `Recomm`). Ce chapitre est également l'occasion d'approfondir la présentation du langage SQL qui n'a été vu que superficiellement jusqu'à présent.

Il existe de nombreuses améliorations possibles au code donné ci-dessous. Quelques-unes sont suggérées au passage. En règle générale, c'est un bon exercice de reprendre ces fonctionnalités et de chercher à les modifier.

7.1 AUTHENTIFICATION

Dans tout site web interactif, on doit pouvoir identifier les internautes avant de leur fournir un accès aux services du site. En ce qui nous concerne, nous avons besoin de savoir qui note les films pour pouvoir faire des prédictions. La procédure classique, dans ce cas, est la suivante :

- lors du premier accès au site, on propose au visiteur de s'inscrire en fournissant un identifiant (pour nous ce sera l'e-mail) et un mot de passe ;
- lors des accès suivants, on lui demande de s'identifier par la paire *(email, mot de passe)*.

7.1.1 Problème et solutions

Comme déjà évoqué à la fin du chapitre 1, le protocole HTTP ne conserve pas d'informations sur la communication entre un programme client et un programme serveur. Si on s'en contentait, il faudrait demander, pour *chaque accès*, un identifiant et un mot de passe, ce qui est clairement inacceptable.

La solution est de créer un ou plusieurs *cookies* pour stocker le nom et le mot de passe du côté du programme client. Rappelons (voir la fin du chapitre 1, page 17) qu'un *cookie* est essentiellement une donnée transmise par le programme serveur au programme client, ce dernier étant chargé de la conserver pour une durée déterminée. Cette durée peut d'ailleurs excéder la durée d'exécution du programme client lui-même, ce qui implique que les cookies soient stockés dans un fichier texte sur la machine cliente.

On peut créer des *cookies* à partir d'une application PHP avec la fonction `SetCookie()`. Il faudrait donc transmettre l'e-mail et le mot de passe après les avoir récupérés par l'intermédiaire d'un formulaire, et les relire à chaque requête d'un programme client. Ce processus est relativement sécurisé puisque seul le programme serveur qui a créé un *cookie* peut y accéder, ce qui garantit qu'un autre serveur ne peut pas s'emparer de ces informations. En revanche toute personne pouvant lire des fichiers sur la machine client peut alors trouver dans le fichier *cookies* la liste des sites visités avec le nom et le mot de passe qui permettent d'y accéder...

Sessions temporaires

La solution la plus sécurisée (ou la moins perméable...) est une variante de la précédente qui fait appel au système de sessions web dont les principes ont été exposés chapitre 2, page 98. Cette variante permet de transmettre le moins d'informations possible au programme client. Elle repose sur l'utilisation d'une base de données du côté serveur et peut être décrite par les étapes suivantes :

1. quand un utilisateur fournit un e-mail et un mot de passe, on compare ces informations à celles stockées dans la base, soit dans notre cas dans la table *Internaute* ;
2. si le nom et le mot de passe sont corrects, on crée une ligne dans une nouvelle table *SessionWeb*, avec un identifiant de session et une durée de validité ;

3. on transmet au client un *cookie* contenant uniquement l'identifiant de session ;
4. si l'identification est incorrecte, on refuse d'insérer une ligne dans *SessionWeb*, et on affiche un message – poli – en informant l'internaute ;
5. à chaque accès du même programme client par la suite, on récupère l'identifiant de session dans le *cookie*, vérifie qu'il correspond à une session toujours valide, et on connaîtra du même coup l'identité de l'internaute qui utilise le site.

Ce processus est un peu plus compliqué, mais il évite de faire voyager sur l'Internet une information sensible comme le mot de passe. Dans le pire des cas, l'identifiant d'une session sera intercepté, avec des conséquences limitées puisqu'il n'a qu'une validité temporaire.

7.1.2 Contrôleur d'authentification et de gestion des sessions

Nous allons ajouter au schéma de la base *Films* une table *SessionWeb* dont voici la description. Comme quelques autres, la commande de création de cette table se trouve dans le script SQL *ComplFilms.sql*.

```
CREATE TABLE SessionWeb (id_session      VARCHAR (40) NOT NULL,
                         email           VARCHAR(60) NOT NULL,
                         nom             VARCHAR(30) NOT NULL,
                         prenom          VARCHAR(30) NOT NULL,
                         temps_limite    DECIMAL (10,0) NOT NULL,
                         PRIMARY KEY (id_session),
                         FOREIGN KEY (email) REFERENCES Internaute
                        );
```

Chaque ligne insérée dans cette table signifie que pour la session `id_session`, l'internaute identifié par `email` à un droit d'accès au site jusqu'à `temps_limite`. Ce dernier attribut est destiné à contenir une date et heure représentées par le nombre de secondes écoulées depuis le premier janvier 1970 (dit « temps UNIX »). Il existe des types spécialisés sous MySQL pour gérer les dates et les horaires, mais cette représentation sous forme d'un entier suffit à nos besoins. Elle offre d'ailleurs le grand avantage d'être comprise aussi bien par MySQL que par PHP, ce qui facilite beaucoup les traitements de dates.

Le nom et le prénom de l'internaute ne sont pas indispensables. On pourrait les trouver dans la table *Internaute* en utilisant l'e-mail. En les copiant dans *SessionWeb* chaque fois qu'une session est ouverte, on évite d'avoir à faire une requête SQL supplémentaire. La duplication d'information est sans impact désagréable ici, puisque les lignes de *SessionWeb* n'existent que temporairement. D'une manière générale cette table, ainsi que d'autres éventuellement créées et référençant l'identifiant de session, peut servir de stockage temporaire pour des données provenant de l'utilisateur pendant la session, comme par exemple le « panier » des commandes à effectuer dans un site de commerce électronique.

Fonctionnalités de gestion des sessions

Les fonctionnalités relatives aux sessions sont placées d'une part dans la super-classe `Controleur`, ce qui permet d'en faire hériter tous les contrôleurs du site, d'autre part dans le contrôleur `Auth` qui se charge de gérer les actions de connexion (*login*) et déconnexion (*logout*). Le site WEBSCOPE est conçu, comme beaucoup d'autres, avec une barre de menu où figure un formulaire de saisie du login et du mot de passe. Quand on valide ce formulaire, on est redirigé vers le contrôleur `Auth` qui se charge alors d'ouvrir une session si les informations fournies sont correctes.

Une fois qu'une internaute a ouvert une session, le formulaire de connexion dans la barre de menu est remplacé par un lien permettant de se déconnecter.

La gestion de session s'appuie sur les fonctions PHP, qui donnent automatiquement un identifiant de session (voir page 98). Rappelons que l'identifiant de la session est transmis du programme serveur au programme client, et réciproquement, tout au long de la durée de la session qui est, par défaut, définie par la durée d'exécution du programme client. La propagation de l'identifiant de session – dont le nom par défaut est `PHPSESSID`, ce qui peut se changer dans le fichier de configuration *php.ini* – repose sur les *cookies* quand c'est possible.

Toutes les autres informations associées à une session doivent être stockées dans la base MySQL pour éviter de recourir à un fichier temporaire. On s'assure ainsi d'un maximum de confidentialité.

Les utilitaires de la classe `Controleur`

Les méthodes suivantes de gestion des sessions se trouvent dans la classe `Controleur`. La première prend en argument l'e-mail et le mot de passe et vérifie que ces informations correspondent bien à un utilisateur du site. Si c'est le cas elle renvoie `true`, sinon `false`. La vérification procède en deux étapes. On recherche d'abord les coordonnées de l'internaute dans la table *Internaute* avec la variable `$email`, puis on compare les mots de passe. Si tout va bien, on crée la session dans la table.

Le mot de passe stocké dans *Internaute* est crypté avec la fonction `md5()` qui renvoie une chaîne de 32 caractères (voir le script d'insertion d'un internaute à la fin du chapitre précédent). Il n'y a pas d'algorithme de *décryptage* de cette chaîne, ce qui garantit que même dans le cas où une personne lirait la table contenant les mots de passe, elle ne pourrait pas sans y consacrer beaucoup d'efforts les obtenir en clair. On doit donc comparer l'attribut `mot_de_passe` de la table avec le cryptage de la variable PHP `$mot_de_passe`.

```
protected function creerSession ($email, $mot_de_passe,
    $id_session)
{
  // Recherchons si l'internaute existe
  $email_propre = $this->bd->prepareChaine($email);
  $requete = "SELECT * FROM Internaute WHERE email='
      $email_propre'";
  $res = $this->bd->execRequete ($requete);
```

```
    $internaute = $this->bd->objetSuivant($res);

    // L'internaute existe-t-il ?
    if (is_object($internaute)) {
      // Vérification du mot de passe
      if ($internaute->mot_de_passe == md5($mot_de_passe)) {
        // Tout va bien. On insère dans la table SessionWeb
        $maintenant = date ("U");
        $temps_limite = $maintenant + self::DUREE_SESSION;
        $email = $this->bd->prepareChaine($email);
        $nom = $this->bd->prepareChaine($internaute->nom);
        $prenom = $this->bd->prepareChaine($internaute->prenom);

        $insSession = "INSERT INTO SessionWeb (id_session, email,
            nom, "
        . "prenom, temps_limite) VALUES ('$id_session', "
        . "'$email','$nom','$prenom', '$temps_limite')";
        $resultat = $this->bd->execRequete ($insSession);

        return true;
      }
      // Mot de passe incorrect !
      else      return false;
    }
    else {
      // L'utilisateur $email est inconnu
      return false;
    }
  }
```

Si les deux tests successifs sont couronnés de succès, on peut créer la session. On dispose de toutes les informations nécessaires pour insérer une ligne dans *SessionWeb* (identifiant, e-mail, nom et prénom), la seule subtilité étant la spécification de la durée de validité.

La fonction PHP `date()` permet d'obtenir la date et l'horaire courants sous de très nombreux formats. En particulier la représentation « UNIX », en secondes depuis le premier janvier 1970, est obtenue avec le format `"U"`. L'expression `date("U")` donne donc le moment où la session est créée, auquel il suffit d'ajouter le nombre de secondes définissant la durée de la session, ici 1 heure=3600 secondes, définie par la constante `DUREE_SESSION` de la classe `Controleur`.

La deuxième méthode vérifie qu'une session existante est valide. Elle prend en argument un objet PHP correspondant à une ligne de la table *SessionWeb*, et compare l'attribut `tempsLimite` à l'instant courant. Si la période de validité est dépassée, on détruit la session.

```
  private function sessionValide ($session)
  {
    // Vérifions que le temps limite n'est pas dépassé
    $maintenant = date ("U");
    if ($session->temps_limite < $maintenant) {
      // Destruction de la session
```

```
        session_destroy();

        $requete  = "DELETE FROM SessionWeb "
        . "WHERE id_session='$session->id_session'";
        $resultat = $this->bd->execRequete ($requete);
        return false;
      }
      else {
        // C'est bon ! On prolonge la session
        $temps_limite = $maintenant + self::DUREE_SESSION;
        $prolonge = "UPDATE SessionWeb SET temps_limite='
            $temps_limite' "
        . " WHERE id_session='$session->id_session' ";
        $this->bd->execRequete ($prolonge);
      }
      return true;
    }
```

Enfin on a besoin d'un formulaire pour identifier les internautes. Bien entendu, on utilise la classe *Formulaire*, et une fonction qui prend en argument le nom du script appelé par le formulaire, et un e-mail par défaut.

```
private function formIdentification($url_auth, $email_defaut=""
    )
{
  // Demande d'identification
  $form = new Formulaire ("post", $url_auth);
  $form->debutTable();
  $form->champTexte("Email", "login_email", "$email_defaut",
      30, 60);
  $form->champMotDePasse ("Passe", "login_password", "", 30);
  $form->champValider ("Identification", "ident");
  $form->finTable();
  return $form->formulaireHTML();
}
```

Nous voilà prêts à créer la méthode contrôlant les accès au site.

Initialisation des sessions

Chaque contrôleur dispose, par héritage de la classe `Controleur`, d'un objet `session` initialisé dans le constructeur par un appel à la méthode `initSession()` (voir le code du constructeur dans le chapitre précédent, page 245). Cette initialisation regarde si une session existe et vérifie qu'elle est valide. Si oui, l'objet `session` est créé, représentant la ligne de *SessionWeb* correspondant à la session stockée. Sinon l'objet `session` reste à `null` et on considère que l'utilisateur n'est pas connecté. La fonction prend en argument l'identifiant de session.

```
protected function initSession ($id_session)
  {
    $requete =  "SELECT * FROM SessionWeb WHERE id_session = '
        $id_session' " ;
```

```
    $resultat = $this->bd->execRequete ($requete);
    $this->session = $this->bd->objetSuivant ($resultat);

    /**
     * On vérifie que la session est toujours valide
     */
    if (is_object($this->session)) {
      // La session existe. Est-elle valide ?
      if (!$this->sessionValide ($this->session)) {
        $this->vue->content = "<b>Votre session n'est pas (ou
            plus) valide.<br/></b>\n";
        $this->session = null;
      }
      else {
        // La session est valide: on place le nom
        // de l'utilisateur dans la vue pour pouvoir l'afficher
        $this->vue->session_nom = $this->session->prenom . " "
        . $this->session->nom;
      }
    }
    // Et on renvoie la session (qui peut être null)
    return $this->session;
  }
```

Dans chaque action d'un contrôleur on peut tester si l'objet `session` existe ou pas. Si non, il faut refuser l'accès aux actions reservées aux utilisateurs connectés. On dispose pour cela de la méthode `controleAcces()` suivante, qui affiche un message de refus d'accès si l'utilisateur n'a pas ouvert de session :

```
function controleAcces()
 {
   if (!is_object($this->session)) {
     $this->vue->contenu = "Vous devez être identifié "
     . "pour accéder à cette page<br/>";
     echo $this->vue->render("page");
     exit;
   }
 }
```

À titre d'exemple voici l'action `index` du contrôleur `Notation`. On commence par appeler `controleAcces()`, et on sait ensuite, si l'action continue à se dérouler, que l'utilisateur est connecté. On dispose même de l'objet `$this->session` pour accéder à ses prénom, nom et e-mail si besoin est.

```
function index ()
 {
   // Définition du titre
   $this->vue->titre_page = "Recherche et notation des films";

   // Contrôle de la session
   $this->controleAcces();

   // Maintenant nous sommes identifiés
```

```
    $this->vue->setFile("contenu", "notation.tpl");

    // Production du formulaire de recherche
    $this->vue->formulaire = $this->formRecherche();
    echo $this->vue->render("page");
  }
```

Finalement on dispose d'une méthode `statutConnexion()` qui se charge de placer dans la vue les informations relatives à la session courante. Deux cas sont possibles :

1. soit la session existe, et on affiche le nom de l'utilisateur connecté, avec un lien de déconnexion ;
2. soit elle n'existe pas, et on affiche le formulaire de connexion.

Cette information est placée dans l'entité `auth_info` de la vue. Notez dans le code l'exploitation des informations de l'objet `$this->session`

```
protected function statutConnexion()
 {
    // S'il n'y a pas de session: on affiche le formulaire
    // d'identification, sinon on place un lien de déconnexion
    if ($this->connexion()) {
      $this->vue->auth_info = "Vous êtes " .
      $this->session->prenom . " " . $this->session->nom . "."
      . "Vous pouvez vous <a href='?ctrl=auth&amp;action=logout'>"
      . "déconnecter</a> à tout moment.";
    }
    else {
      $this->vue->auth_info =
          $this->FormIdentification("?ctrl=auth&amp;action=login");
    }
  }
```

7.1.3 Les actions de *login* et de *logout*

Ces deux actions font partie du contrôleur `Auth`. L'action *login* reçoit un em-ail et un mot de passe (qu'il faut vérifier) et tente de créer une session avec les utilitaires de gestion de session hérités de la classe `Controleur`.

```
function login ()
 {
    $this->titre_page = "Identification";

    // Si on est déjà connecté: on refuse
    if (is_object($this->session)) {
      $this->vue->contenu = "Vous êtes déjà connecté. Déconectez-
          vous "
      . " au préalable avant de choisir un autre compte.";
    }
```

```
    else if (isSet($_POST['login_email']) and
      isSet($_POST['login_password']) ) {
      // Une paire email/mot de passe existe. Est-elle correcte ?

      if ($this->creerSession ($_POST['login_email'],
              $_POST['login_password'], session_id())) {
        // On initialise l'objet session avec les données qu'on
        // vient de créer
        $this->initSession (session_id());
        // Affichage d'une page d'accueil sympathique
        $this->vue->setFile ("contenu", "auth_login.tpl");
      }
      else
        $this->vue->contenu .= "<center><b>Votre identification
            a échoué.</b></center>\n";
    }
    else {
      $this->vue->contenu = "Vous devez fournir votre email et
          votre mot de passe<br/>";
    }

    // Rafraichissement de la partie du contenu qui montre soit
    // un formulaire, de connexion, soit un lien de déconnexion
    $this->statutConnexion();

    echo $this->vue->render("page");
  }
```

La figure 7.1 montre la présentation du site juste après l'identification d'un internaute. Outre le message d'accueil dans la partie centrale, on peut noter que le statut de connexion affiché dans le menu à droite montre maintenant les prénom et nom de l'internaute connecté, ainsi qu'un lien qui pointe vers l'action de déconnexion `logout`.

Cette dernière vérifie que l'internaute est bien connecté (autrement dit, que l'objet `session` existe) et effectue alors une destruction dans la table, ainsi que par appel à la fonction PHP `session_destroy()`. L'objet `session` est également remis à `null` et le bloc d'information dans la vue réinitialisé.

```
  public function logout ()
  {
    $this->vue->titre_page = "Déconnexion";

    // Vérifions qu'on est bien connecté
    if (is_object($this->session))
    {
      $this->vue->contenu = "Vous étiez identifié sous le nom "
      . "<b>{$this->session->prenom} {$this->session->nom}</b>
         <br/>";
      session_destroy();
```

```
        $requete  = "DELETE FROM SessionWeb "
          . " WHERE id_session='{$this->session->id_session}'";
        $resultat = $this->bd->execRequete ($requete);
        $this->session = null;

        $this->vue->contenu .= "Vous êtes maintenant déconnecté !\n";
      }
      else
        $this->vue->contenu = "Vous n'êtes pas encore connecté !\n";

       // Rafraichissement de la partie du contenu qui montre soit
      // un formulaire de connexion, soit un lien de déconnexion
      $this->statutConnexion();

      echo $this->vue->render("page");
    }
```

Il se peut que `logout()` ne soit pas appelé par un internaute qui a simplement quitté le site sans passer par `logout`, et que l'information sur la session, bien que devenue invalide, reste dans la base. On peut au choix la garder à des fins statistiques, ou nettoyer régulièrement les sessions obsolètes.

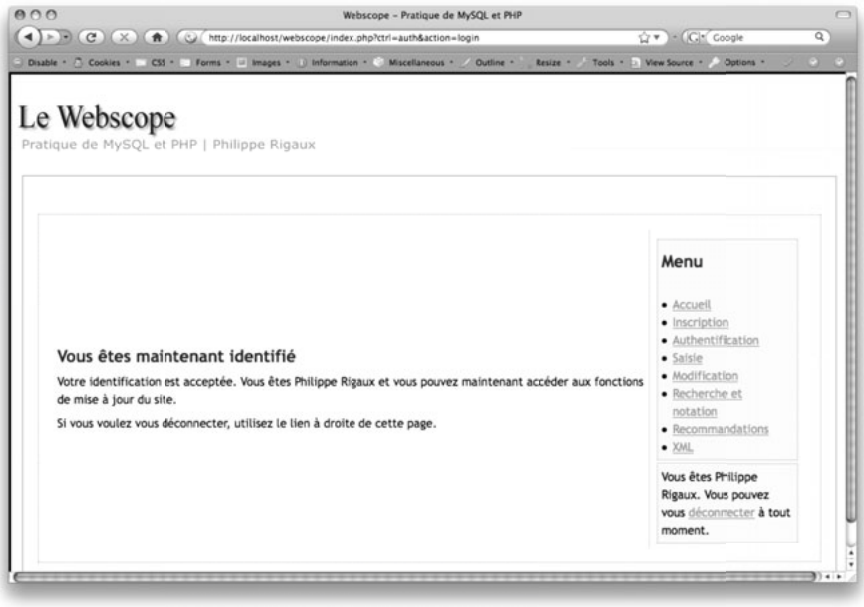

Webscope – Pratique de MySQL et PHP
http://localhost/webscope/index.php?ctrl=auth&action=login
Le Webscope
Pratique de MySQL et PHP | Philippe Rigaux
Vous êtes maintenant identifié
Votre identification est acceptée. Vous êtes Philippe Rigaux et vous pouvez maintenant accéder aux fonctions de mise à jour du site.
Si vous voulez vous déconnecter, utilisez le lien à droite de cette page.
Menu
Accueil
Inscription
Authentification
Saisie
Modification
Recherche et notation
Recommandations
XML
Vous êtes Philippe Rigaux. Vous pouvez vous déconnecter à tout moment.

Figure 7.1 — Page d'accueil après identification d'un internaute

7.2 RECHERCHE, PRÉSENTATION, NOTATION DES FILMS

Nous en arrivons maintenant aux fonctionnalités principales du site WEBSCOPE, à savoir rechercher des films, les noter et obtenir des recommandations. Tout ce qui suit fait partie du contrôleur `Notation`. Nous utilisons explicitement des requêtes SQL pour simplifier l'exposé, une amélioration possible étant de suivre scrupuleusement le MVC en définissant des modèles.

7.2.1 Outil de recherche et jointures SQL

La recherche repose sur un formulaire, affiché dans la figure 7.2, qui permet d'entrer des critères de recherche. Ces critères sont :

- le titre du film ;
- le nom d'un metteur en scène ;
- le nom d'un acteur ;
- le genre du film ;
- un intervalle d'années de sortie.

Webscope – Pratique de MySQL et PHP
Recherche et notation des films
Le formulaire ci-dessous vous permet d'effectuer une recherche en fonction d'un ensemble de critères. Cette recherche est basée sur les principes suivants :
1. Vous pouvez saisir, en majuscules ou minuscules, tout ou partie du contenu d'un champ.
2. Un champ non-saisi est "neutre" : il n'est pas pris en compte pour la recherche.
Exemple : la valeur ver dans le champ Titre, et 1950 dans le champ Année min. indiquent une recherche de tous les films contenant la chaîne 'ver' et produit après 1950.
Titre du film
Metteur en scène
Acteur
Tous
Genre
Année min.
Année max.
Action
Aventures
1800
2100
Rechercher
Menu
Accueil
Inscription
Authentification
Saisie
Modification
Recherche et notation
Recommandations
XML
Vous êtes Philippe Rigaux. Vous pouvez vous déconnecter à tout moment.

Figure 7.2 — Formulaire de recherche des films

Les quatre premiers champs ont chacun comme valeur par défaut « Tous », et l'intervalle de date est fixé par défaut à une période suffisamment large pour englober tous les films parus. De plus, on accepte une spécification partielle du titre ou des noms. Si un internaute entre « ver », on s'engage à rechercher tous les films contenant cette chaîne.

Voici le formulaire, produit avec la classe `Formulaire` dans le cadre d'une méthode privée du contrôleur `Notation`. On aurait pu aussi créer un *template* avec

le code HTML et y injecter la liste des genres, qui est la seule partie dynamique provenant de la base.

Les champs sont groupés par trois, et affichés dans deux tableaux en mode horizontal.

```
private function formRecherche ()
 {
   // Recherche de la liste des genres
   $resultat  = $this->bd->execRequete("SELECT code FROM Genre");
   $genres["Tous"] = "Tous";
   while ($g=$this->bd->objetSuivant($resultat)) $genres[$g->
       code] = $g->code;

   // Création du formulaire
   $form = new Formulaire ("POST", "?ctrl=notation&amp;action=
       recherche");

   $form->debutTable(Formulaire::HORIZONTAL);
   $form->champTexte ("Titre du film", "titre", "Tous", 20);
   $form->champTexte ("Metteur en scène", "nom_realisateur",
       "Tous", 20);
   $form->champTexte("Acteur", "nom_acteur", "Tous", 20);
   $form->finTable();
   $form->ajoutTexte("<br/>");
   $form->debutTable(Formulaire::HORIZONTAL);
   $form->champListe ("Genre", "genre", "Tous", 3, $genres);
   $form->champTexte ("Année min.", "annee_min", 1800, 4);
   $form->champTexte ("Année max.", "annee_max", 2100, 4);
   $form->finTable();

   $form->champValider ("Rechercher","rechercher");
   return $form->formulaireHTML();
 }
```

Requête sur une table

Dans le cas où les champs `nom_realisateur` ou `nom_acteur` restent à « Tous », il ne faut pas les prendre en compte. Les critères de recherche restant font tous référence à des informations de la table *Film*. On peut alors se contenter d'une requête SQL portant sur une seule table, et utiliser la commande `LIKE` pour faire des recherches sur une partie des chaînes de caractères (voir Exemple 1.10, page 43). Voici la requête que l'on peut utiliser.

```
SELECT titre, annee, code_pays, genre, id_realisateur
FROM Film
WHERE titre LIKE '%$titre%'
AND annee BETWEEN '$annee_min' AND '$annee_max'
AND genre LIKE '$genre'
```

Jointures

Supposons maintenant que la variable `$nom_realisateur` ne soit pas égale à « Tous ». Il faut alors tenir compte du critère de sélection sur le nom du metteur en scène pour sélectionner les films et on se retrouve face à un problème pas encore abordé jusqu'à présent : effectuer des ordres SQL impliquant plusieurs tables.

SQL sait très bien faire cela, à condition de disposer d'un moyen pour rapprocher une ligne de la table *Film* de *la* (l'unique) ligne de la table *Artiste* qui contient les informations sur le metteur en scène. Ce moyen existe : c'est l'attribut `id_realisateur` de *Film* qui correspond à la clé de la table *Artiste*, `id`. Rapprocher les films de leur metteur en scène consiste donc, pour une ligne dans *Film*, à prendre la valeur de `id_realisateur` et à rechercher dans *Artiste* la ligne portant cet `id`. Voici comment on l'exprime avec SQL.

```
SELECT titre, annee, code_pays, genre, id_realisateur
FROM   Film, Artiste
WHERE  titre LIKE '%$titre%'
AND    nom LIKE '%$nom_realisateur%'
AND    annee BETWEEN $annee_min AND $annee_max
AND    genre LIKE '$genre'
AND    id_realisateur = Artiste.id
```

Ce type d'opération, joignant plusieurs tables, est désigné par le terme de *jointure*. La syntaxe reste identique, avec une succession de clauses `SELECT-FROM-WHERE`, mais le `FROM` fait maintenant référence aux deux tables qui nous intéressent, et le critère de rapprochement de lignes venant de ces deux tables est indiqué par l'égalité `id_realisateur = id` dans la clause `WHERE`.

Les attributs auxquels on peut faire référence, aussi bien dans la clause `WHERE` que dans la clause `SELECT`, sont ceux de la table *Film et* de la table *Artiste*. Dans ce premier exemple, tous les attributs ont des noms différents et qu'il n'y a donc aucune ambiguïté à utiliser l'attribut `nom` ou `annee` sans dire de quelle table il s'agit. MySQL sait s'y retrouver.

Prenons maintenant le cas où la variable `$nom_realisateur` est égale à « Tous », tandis qu'un critère de sélection des acteurs a été spécifié. Le cas est un peu plus complexe car pour rapprocher la table *Film* de la table *Artiste*, il faut impliquer également la table *Role* qui sert d'intermédiaire (voir chapitre 4, page 195).

Voici la requête SQL effectuant la jointure.

```
SELECT Film.titre, annee, code_pays, genre, id_realisateur
FROM Film, Artiste, Role
WHERE Film.titre LIKE '%$titre%'
AND  nom LIKE '%$nom_acteur%'
AND  annee BETWEEN '$annee_min' AND '$annee_max'
AND  genre LIKE '$genre'
AND  id_acteur = Artiste.id
AND  Role.id_film = Film.id
```

Comme dans le cas de la jointure entre *Film* et *Artiste* pour rechercher le metteur en scène, la jointure entre ces trois tables se fonde sur les attributs communs qui sont :

1. les attributs `id` et `id_film` dans *Film* et *Role* ;
2. les attributs `id` et `id_acteur` dans, respectivement, *Acteur* et *Role*.

Il y a ambiguïté sur `id` puisque MySQL ne peut pas déterminer, quand on utilise cet attribut, si on fait référence à *Film* ou à *Artiste*. Pour lever cette ambiguïté, on préfixe donc le nom de l'attribut par le nom de la table d'où il provient.

Dans le cas le plus général, l'utilisateur entre une valeur pour le metteur en scène et une pour le nom de l'acteur. En indiquant par exemple « itch » dans le champ `nom_realisateur` et « ewa » dans le champ `nom_acteur`, on devrait obtenir (au moins) le film *Vertigo*, dirigé par Alfred H*itch*cock, et joué par James St*ewa*rt.

Il faut donc à la fois faire la jointure *Film-Artiste* pour le metteur en scène, et *Film-Role-Artiste* pour les acteurs. On recherche en fait, simultanément, *deux* lignes dans la table *Artiste*, l'une correspondant au metteur en scène, l'autre à l'acteur. Tout se passe comme si on effectuait une recherche d'une part dans une table contenant tous les acteurs, d'autre part dans une table contenant tous les metteurs en scène.

C'est exactement ainsi que la requête SQL doit être construite. On utilise deux fois la table *Artiste* dans la clause `FROM`, et on la renomme une fois en *Acteur*, l'autre fois en *MES* avec la commande SQL `AS`. Ensuite on utilise le nom approprié pour lever les ambiguïtés quand c'est nécessaire.

```
SELECT Film.titre, annee, code_pays, genre, id_realisateur
FROM Film, Artiste AS Acteur, Artiste AS MES, Role
WHERE Film.titre LIKE '%$titre%'
AND Acteur.nom LIKE '%$nom_acteur%'
AND MES.nom LIKE '%$nom_realisateur%'
AND annee BETWEEN '$annee_min' AND '$annee_max'
AND genre LIKE '$genre'
AND id_acteur = Acteur.id
AND id_realisateur = MES.id
AND Role.id_film = Film.id
```

Cette requête est d'un niveau de complexité respectable, même si on peut aller plus loin. Une manière de bien l'interpréter est de raisonner de la manière suivante.

L'exécution d'une requête SQL consiste à examiner *toutes* les combinaisons possibles de lignes provenant de *toutes* les tables de la clause `FROM`. On peut alors considérer chaque nom de table dans le `FROM` comme une variable qui pointe sur une des lignes de la table. Dans l'exemple ci-dessus, on a donc deux variables, `Acteur` et `MES` qui pointent sur deux lignes de la table *Artiste*, et deux autres, `Film` et `Role` qui pointent respectivement sur des lignes des tables *Film* et *Role*.

Étant données les lignes référencées par ces variables, la clause `SELECT` renvoie un résultat si tous les critères de la clause `WHERE` sont vrais simultanément. Le résultat est lui-même construit en prenant un ensemble d'attributs parmi ces lignes.

Nous reviendrons plus systématiquement sur les possibilités du langage SQL dans le chapitre 10. Voici, pour conclure cette section, la méthode `creerRequetes()` qui initialise, en fonction des saisies de l'internaute, la requête à exécuter.

Cette méthode est un peu particulière. Il ne s'agit pas vraiment d'une méthode au sens habituel de la programmation objet, puisqu'elle ne travaille pas dans le contexte d'un objet et se contente de faire de la manipulation syntaxique pour produire une chaîne de caractères contenant une requête SQL. Dans ce cas on peut la déclarer comme une méthode *statique*. Une méthode statique (ou méthode de classe) ne s'exécute pas dans le contexte d'un objet ; on ne peut donc pas y faire référence à `$this`. On ne peut pas non plus appeler une méthode statique avec la syntaxe « `$this->` ». L'appel se fait donc en préfixant le nom de la méthode par le nom de sa classe :

`NomClasse``::`*`nomMéthode`*

Les méthodes statiques sont souvent utilisées pour fournir des services généraux à une clase, comme compter le nombre d'objets instanciés depuis le début de l'exécution. On pourrait placer `creerRequetes()` comme une méthode statique du contrôleur `Notation`. Ici, il faut bien réfléchir à ce qu'est un contrôleur : un conteneur d'actions déclenchées par des requêtes HTTP. Si on commence à placer des méthodes fonctionnelles, autres que des actions, dans un contrôleur, on ne pourra pas les utiliser ailleurs. Nous aurons besoin de `creerRequetes()` dans d'autres contrôleurs. Il ne reste donc plus qu'à créer une classe spécifiquement dédiée aux fonctions utilitaires qui ne sont ni des actions, ni des méthodes d'un modèle. Vous trouverez dans le répertoire *application/classes* une classe `Util` qui ne contient que des méthodes statiques tenant lieu d'utilitaires pour l'application. On y trouve par exemple des fonctions cherchant des lignes dans la table *Artiste*, par clé, par prénom et nom, et quelques autres que nous présenterons ensuite. On y trouve donc la méthode `creerRequetes()`.

```
static function creerRequetes ($tab_criteres, $bd)
 {
   // On décode les critères en les préparant pour
   // la requête SQL. Quelques tests seraient bienvenus.

   if ($tab_criteres['titre'] == "Tous")  $titre = '%';
   else $titre = $bd->prepareChaine($tab_criteres['titre']);

   if ($tab_criteres['genre'] == "Tous") $genre = '%';
   else  $genre = $bd->prepareChaine($tab_criteres['genre']);

   if ($tab_criteres['nom_realisateur'] == "Tous")
   $nom_realisateur = '%';
   else  $nom_realisateur =
   $bd->prepareChaine($tab_criteres['nom_realisateur']);

   if ($tab_criteres['nom_acteur'] == "Tous") $nom_acteur = '%';
   else  $nom_acteur =
   $bd->prepareChaine($tab_criteres['nom_acteur']);
```

```
    $annee_min = $tab_criteres['annee_min'];
    $annee_max = $tab_criteres['annee_max'];

    // Maintenant on construit la requête
    if ($nom_realisateur == "%" and $nom_acteur == "%")
    {
      // Une requête sur la table Film suffit
      $requete = "SELECT * FROM Film "
      . "WHERE titre LIKE '%$titre%' "
      . "AND annee BETWEEN '$annee_min' AND '$annee_max' "
      . "AND genre LIKE '$genre' ";
    }
    else if ($nom_acteur == "%")
    {
      // Il faut une jointure Film-Artiste
      $requete = "SELECT Film.* "
      . "FROM Film, Artiste "
      . "WHERE titre LIKE '%$titre%' "
      . "AND  nom LIKE '%$nom_realisateur%' "
      . "AND  annee BETWEEN '$annee_min' AND '$annee_max' "
      . "AND genre LIKE '$genre' "
      . "AND  id_realisateur = Artiste.id ";
    }
    else if ($nom_realisateur == "%")
    {
      // Il faut une jointure Film-Artiste-Role
      $requete = "SELECT Film.* "
      . "FROM Film, Artiste, Role "
      . "WHERE Film.titre LIKE '%$titre%' "
      . "AND  nom LIKE '%$nom_acteur%' "
      . "AND  annee BETWEEN '$annee_min' AND '$annee_max' "
      . "AND  genre LIKE '$genre' "
      . "AND  id_acteur = Artiste.id "
      . "AND  Role.id_film = Film.id ";
    }
    else
    {
      // On construit la requête la plus générale
      $requete = "SELECT Film.* "
      . "FROM Film, Artiste AS Acteur, Artiste AS MES, Role "
      . "WHERE Film.titre LIKE '%$titre%' "
      . "AND  Acteur.nom LIKE '%$nom_acteur%' "
      . "AND  MES.nom LIKE '%$nom_realisateur%' "
      . "AND  annee BETWEEN '$annee_min' AND '$annee_max' "
      . "AND  genre LIKE '$genre' "
      . "AND  id_acteur = Acteur.id "
      . "AND  id_realisateur = MES.id "
      . "AND  Role.id_film = Film.id_film ";
    }
    return $requete;
}
```

Le regroupement de méthodes statiques dans une classe dédiée est très proche de la notion de bibliothèque de fonctions. La différence tient d'une part à la structuration du code, plus forte en programmation objet (concrètement, on trouve facilement d'où vient une méthode statique puisqu'elle est toujours accolée au nom de sa classe), et d'autre part à la présence des propriétés (variables) statiques alors communes à toutes les méthodes statiques d'une classe.

7.2.2 Notation des films

Au moment de l'exécution d'une recherche, une des requêtes précédentes – celle correspondant à la demande de l'utilisateur – est créée par `creerRequetes()`. On l'exécute alors, et on présente les films obtenus dans un tableau, lui-même inclus dans un formulaire. Ce tableau comprend deux colonnes (figure 7.3).

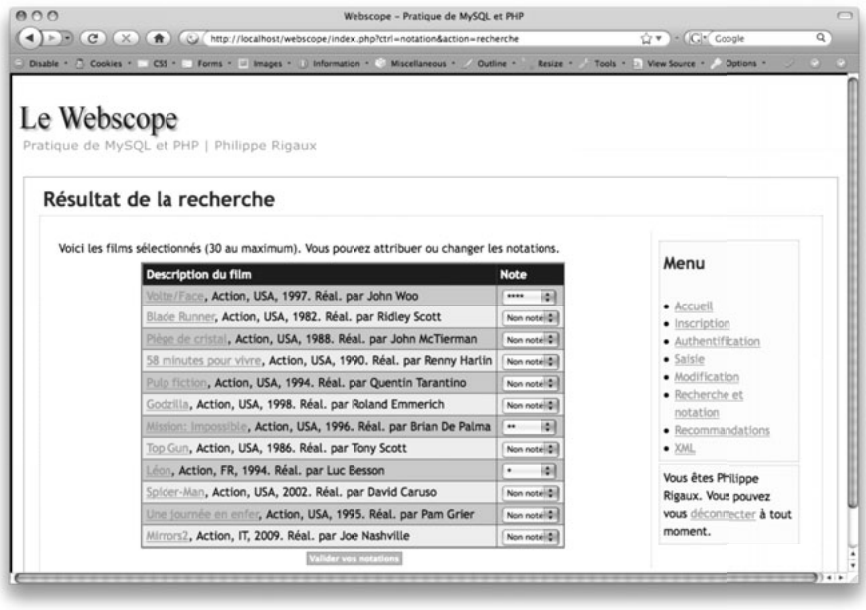

Webscope – Pratique de MySQL et PHP
http://localhost/webscope/index.php?ctrl=notation&action=recherche
Le Webscope
Pratique de MySQL et PHP | Philippe Rigaux
Résultat de la recherche
Voici les films sélectionnés (30 au maximum). Vous pouvez attribuer ou changer les notations.
Description du film
Note
Volte/Face, Action, USA, 1997. Réal. par John Woo
Blade Runner, Action, USA, 1982. Réal. par Ridley Scott
Piège de cristal, Action, USA, 1988. Réal. par John McTiernan
58 minutes pour vivre, Action, USA, 1990. Réal. par Renny Harlin
Pulp fiction, Action, USA, 1994. Réal. par Quentin Tarantino
Godzilla, Action, USA, 1998. Réal. par Roland Emmerich
Mission: Impossible, Action, USA, 1996. Réal. par Brian De Palma
Top Gun, Action, USA, 1986. Réal. par Tony Scott
Léon, Action, FR, 1994. Réal. par Luc Besson
Spicer-Man, Action, USA, 2002. Réal. par David Caruso
Une journée en enfer, Action, USA, 1995. Réal. par Pam Grier
Mirrors2, Action, IT, 2009. Réal. par Joe Nashville
Valider vos notations
Menu
Accueil
Inscription
Authentification
Saisie
Modification
Recherche et notation
Recommandations
XML
Vous êtes Philippe Rigaux. Vous pouvez vous déconnecter à tout moment.

Figure 7.3 — Formulaire de notation des films

1. La première colonne contient une présentation des films, avec le titre, le genre, l'année, etc. Le titre est une ancre vers le contrôleur `Film` qui donne un affichage complet d'un film, incluant son affiche et son résumé. L'URL associée à cette ancre est

   ?ctrl=film&action=index&id_film={id_film}

2. La deuxième colonne est un champ de type liste, proposant les notes, avec comme valeur par défaut la note déjà attribuée par l'internaute à un film, si elle existe. Si la note n'existe pas, elle est considérée comme valant 0 ce qui correspond à l'intitulé « Non noté » dans le tableau `$liste_notes`. De plus, on place dans cette colonne un champ caché, pour chaque ligne, contenant le titre du film.

La recherche est implantée comme une action MVC combinant de manière classique un code PHP accédant à la base, et une vue définie par un *template*. Voici tout d'abord la vue.

Exemple 7.1 *La vue pour l'affichage des films à noter*

```
 <p>
 Voici les films sélectionnés ({max_films} au maximum).
  Vous pouvez attribuer ou changer les notations.<br/>
</p>

<center>
<form  method='post' action='?ctrl=notation&amp;action=noter'
        name='Form'><table  border='2' >
<tr class='header'><th>Description du film</th><th>Note</th></tr>

<!-- BEGIN film -->
<tr bgcolor='Silver' >

<!-- Champ caché avec l'identifiant du film -->
<input type='hidden' name="id[]" value="{id_film}"/>

<!-- Ancre pour voir la description du film -->
<td><a href='?ctrl=film&amp;action=index&amp;id_film={id_film}'>{
    titre}</a>,
         {genre}, {pays}, {annee},  Réal. par {realisateur}</td>

<!-- Champ select pour attribuer une note -->
<td>{liste_notes}</td>

</tr>
<!-- END film -->
</table>

<input type='submit' name="valider" value="Valider vos notations"/>
</form>
</center>
```

Une ligne du tableau d'affichage est représentée par une *template* imbriqué nommé `film`. On instancie ce template autant de fois qu'on trouve de films dans le résultat de la requête basée sur les critères saisis par l'utilisateur. Voici l'action du contrôleur `Notation`.

```
  function recherche ()
  {
    // Contrôle de la session
    $this->controleAcces();

    // Définition du titre et de la vue
    $this->vue->titre_page = "Résultat de la recherche";

    $this->vue->setFile("contenu", "notation_recherche.tpl");
```

```
// Extraction du template 'film', remplacement par l'entité
// 'films'
$this->vue->setBlock("contenu", "film", "films");

// Création de la liste des notes
$notes = array ("0"=>"Non noté", "1"=>'*',  "2"=>'**',
                "3"=>'***', "4"=>'****',  "5"=>'*****');

// Création de la requête en fonction des critères passés au
// script
$requete = Util::creerRequetes ($_POST, $this->bd);
$resultat = $this->bd->execRequete ($requete);
$nb_films=1;
while ($film  = $this->bd->objetSuivant ($resultat)) {
  // Recherche du metteur en scène
  $mes = Util::chercheArtisteAvecID ($film->id_realisateur,
      $this->bd);

  // Placement des informations dans la vue
  $this->vue->id_film = $film->id;
  $this->vue->genre = $film->genre;
  $this->vue->titre = $film->titre;
  $this->vue->annee = $film->annee;
  $this->vue->pays = $film->code_pays;
  $this->vue->realisateur = $mes->prenom . " " . $mes->nom;

  // Recherche de la notation de l'utilisateur courant pour
  // l'utiliser
  // comme valeur par défaut dans le champ de formulaire de
  // type liste
  $notation = Util::chercheNotation ($this->session->email,
      $film->id, $this->bd);
  if (is_object($notation)) {
    $note_defaut = $notation->note;
  }
  else {
    $note_defaut = 0;
  }

  // La liste des notes est un champ <select> créé par une
  // méthode statique de la vue.
  $this->vue->liste_notes =
    Template::champSelect("notes[$film->id]", $notes,
        $note_defaut);

  // Instanciation du template 'film', ajout dans l'entité
  // 'films'
  $this->vue->append("films", "film");

  if ($nb_films++ >= self::MAX_FILMS) break;
}
```

```
    // Finalement on affiche la vue comme d'habitude
    $this->vue->max_films = self::MAX_FILMS;
    echo $this->vue->render("page");
}
```

On effectue une boucle classique sur le résultat de la requête. Chaque passage dans la boucle correspond à une ligne dans le tableau, avec la description du film et l'affichage d'une liste de notes. Pour cette dernière, on se retrouve face au problème classique d'engendrer un champ `<select>` avec une liste d'options, qui constitue une imbrication très dense de valeurs dynamiques (les codes des options) et de textes statiques (les balises HTML). La solution adoptée ici est de s'appuyer sur une fonction utilitaire, implantée comme une méthode statique de la classe `Template`, qui produit le texte HTML à partir d'un tableau PHP.

```
static function  champSelect ($nom, $liste, $defaut, $taille=1)
{
    $options = "";
    foreach ($liste as $val => $libelle) {
        // Attention aux problèmes d'affichage
        $val = htmlSpecialChars($val);
        $defaut = htmlSpecialChars($defaut);

        if ($val != $defaut) {
            $options .= "<option value=\"$val\">$libelle</option>\n";
        }
        else {
            $options .= "<option value=\"$val\" selected='1'>
                $libelle</option>\n";
        }
    }
    return "<select name='$nom' size='$taille'>" . $options . "</
        select>\n";
}
```

Le script associé à ce formulaire reçoit donc deux tableaux PHP : d'abord `$id`, contenant la liste des identifiants de film ayant reçu une notation, et `$notes`, contenant les notes elles-mêmes. Si l'on constate que la note a changé pour un film, on exécute un `UPDATE`, et si la note n'existe pas on exécute un `INSERT`. C'est l'action `noter` qui se charge de cette prise en compte des notations.

```
function noter ()
{
    $this->controleAcces();

    // Définition du titre et de la vue
    $this->vue->titre_page = "Résultat de la notation";
    $this->vue->setFile("contenu", "notation_noter.tpl");

    // Boucle sur tous les films notés
    foreach ($_POST['id']  as $id)  {
        $note = $_POST['notes'][$id];
        $notation = Util::chercheNotation ($this->session->email,
```

```
                $id, $this->bd);

        // On met à jour si la note a changé
        if (!is_object($notation) && $note != 0) {
          $requete = "INSERT INTO Notation (id_film, email, note) "
          . " VALUES ('$id', '{$this->session->email}', '$note')";
          $this->bd->execRequete ($requete);
        }
        else if (is_object($notation) && $note != $notation->note)
            {
          $requete = "UPDATE Notation SET note='$note' "
          . " WHERE email='{$this->session->email}' AND id_film =
              '$id'";
          $this->bd->execRequete ($requete);
        }
      }

      // Production du formulaire de recherche
      $this->vue->formulaire = $this->formRecherche();
      echo $this->vue->render("page");
    }
```

7.3 AFFICHAGE DES FILMS ET FORUM DE DISCUSSION

Le contrôleur `Film` affiche toutes les informations connues sur un film, et propose un forum de discussion (une liste de messages permettant de le commenter). La présentation d'un film (voir l'exemple de la figure 7.4) est une mise en forme HTML des informations extraites de la base *via* PHP. Il s'agit d'un nouvel exemple de production d'une page par templates.

```
function index ()
 {
   // Définition du titre
   $this->vue->titre_page = "Affichage des films";

   // Contrôle de la session
   $this->controleAcces();

   // On devrait avoir reçu un identifiant
   if (!isSet($_REQUEST['id_film'])) {
     $this->vue->contenu = "Je ne peux pas afficher cette page:
         "
     . " il me faut un identifiant de film";
     echo $this->vue->render("page");
     exit;
   }

   // On recherche le film avec l'id
   $film = Util::chercheFilm($_REQUEST['id_film'], $this->bd);
```

```
    // Si on a trouvé le film, on y va !
    if (is_object($film)) {
      $this->vue->setFile("contenu", "film.tpl");
      // Extraction du bloc des acteurs
      $this->vue->setBlock("contenu", "acteur", "acteurs");
      $this->vue->setBlock("contenu", "message", "messages");

      // Il suffit de placer dans la vue les informations
      // nécessaires à l'affichage du film
      $this->vue->id_film = $film->id;
      $this->vue->titre = $film->titre;
      $this->vue->genre = $film->genre;
      $this->vue->pays = $film->code_pays;
      $this->vue->annee = $film->annee;
      $this->vue->resume = $film->resume;
      $this->vue->affiche = "./affiches/" . md5($film->titre) .
          ".gif";

      // La moyenne des notes
      $this->vue->moyenne = Util::moyenneFilm ($film->id, $this->
          bd);

      // On prend le réalisateur
      $mes = Util::chercheArtisteAvecId ($film->id_realisateur,
          $this->bd);
      $this->vue->realisateur_prenom = $mes->prenom;
      $this->vue->realisateur_nom = $mes->nom;

      // Les acteurs
      $requete = "SELECT nom, prenom, nom_role FROM Artiste a,
          Role r "
      . "WHERE a.id = r.id_acteur AND r.id_film='$film->id'";
      $resultat = $this->bd->execRequete ($requete);

      while ($acteur = $this->bd->objetSuivant ($resultat)) {
        $this->vue->acteur_nom = $acteur->nom;
        $this->vue->acteur_prenom = $acteur->prenom;
        $this->vue->acteur_role = $acteur->nom_role;
        $this->vue->append("acteurs", "acteur");
      }

      // Les messages sur le film
      $this->vue->messages = $this->afficheMess ($film->id, 0);

      echo $this->vue->render("page");
    }
  }
```

La seule nouveauté notable est la gestion d'un forum de discussion. Cette idée simple se rencontre couramment : il s'agit d'offrir aux internautes un moyen de déposer des commentaires ou des appréciations, sous la forme d'un message stocké

dans la base MySQL. De plus, afin de rendre possible une véritable discussion, on donne la possibilité de *répondre* à des messages déjà entrés.

Webscope – Pratique de MySQL et PHP
Affichage des films
Vertigo
Drame, USA, 1958
Mis en scène par Alfred Hitchcock
Avec :
James Stewart dans le rôle de John Ferguson
Kim Novak dans le rôle de Madeleine Elster
Résumé
Scottie Ferguson, ancien inspecteur de police, est sujet au vertige depuis qu'il a vu mourir son collègue. Elster, son ami, le charge de surveiller sa femme, Madeleine, ayant des tendances suicidaires. Amoureux de la jeune femme Scottie ne remarque pas le piège qui se trame autour de lui et dont il va être la victime...
Moyenne des notes: 3.0000
Commentaires sur ce film
Ajouter un message sur ce film
Menu
Accueil
Inscription
Authentification
Saisie
Modification
Recherche et notation
Recommandations
XML
Vous êtes Philippe Rigaux. Vous pouvez vous déconnecter à tout moment.

Figure 7.4 — Présentation d'un film

Les messages constituent donc une hiérarchie, un message étant fils d'un autre s'il lui répond. La table stockant les messages doit contenir l'identifiant du film, l'e-mail de l'internaute qui a saisi le message, l'identifiant éventuel du message dont il est le fils, et bien entendu le sujet, le texte et la date de création du commentaire. Voici le script de création de la table, qui se trouve dans le fichier *ComplFilms.sql*.

```
CREATE TABLE Message (id INT NOT NULL,
                      id_pere INT DEFAULT 0,
                      id_film  INTEGER (50) NOT NULL,
                      sujet  VARCHAR (30) NOT NULL,
                      texte TEXT NOT NULL,
                      date_creation INT,
                      email_createur  VARCHAR(40) NOT NULL,
                      PRIMARY KEY (id),
                      FOREIGN KEY (email_createur) REFERENCES
                          Internaute);
```

Les messages de plus haut niveau (ceux qui ne constituent pas une réponse) auront un `id_pere` nul, comme l'indique la clause `DEFAULT`.

La saisie des messages s'effectue dans un formulaire produit par l'action `message` du contrôleur `Film`. Cette action s'attend à recevoir le titre du film, et éventuellement l'identifiant du message auquel on répond. Dans ce dernier cas on n'affiche pas le sujet qui reprend celui du message-père, et qui est inclus dans un champ caché.

```
function message()
{
  // Définition du titre
  $this->vue->titre_page = "Ajout d'un message";
```

```
    // Contrôle de la session
    $this->controleAcces();

    // Vérification des valeurs passées au script
    if (empty($_REQUEST['id_film'])) {
      $this->vue->contenu = "<b>Il manque des informations ?!
          </b>\n";
    }
    else {
      // Ce message est-il le fils d'un autre message ?
      if (!isSet($_REQUEST['id_pere'])) {
        $id_pere = "";
      }
      else {
        $id_pere = $_REQUEST['id_pere'];
      }

      // Création du formulaire
      $f = new Formulaire("post", "?ctrl=film&amp;action=inserer");

      // Champs cachés: email, titre du film, id du message père
      $f->champCache ("email_createur", $this->session->email);
      $f->champCache ("id_film", $_REQUEST['id_film']);
      $f->champCache ("id_pere", $id_pere);

      // Tableau en mode vertical
      $f->debutTable();
      // S'il s'agit d'une réponse: on n'affiche pas le sujet
      if ($id_pere == "" or $id_pere == 0)      {
        $f->champTexte ("Sujet", "sujet", "", 30);
      }
      else {
        $f->ajoutTexte ("<h3>Réponse au message <i>"
              . $_REQUEST['sujet'] . "</i></h3>\n");
        $f->champCache ("sujet", $_REQUEST['sujet']);
      }
      $f->champFenetre ("Message", "texte", "", 4, 50);
      $f->finTable();

      $f->champValider ("Enregistrer le message", "valider");

      // Affichage du formulaire
      $this->vue->formulaire = $f->formulaireHTML();

      $this->vue->id_film = $_REQUEST['id_film'];
      $this->vue->email_createur = $this->session->email;
      $this->vue->id_pere = $id_pere;
      $this->vue->setFile("contenu", "film_message.tpl");
    }
    echo $this->vue->render("page");
}
```

Nous ne donnons pas le code d'insertion des messages similaire à ceux déjà vus pour les films ou les internautes. Vous pouvez le trouver dans le code de *FilmCtrl.php*. En revanche, il est plus intéressant d'examiner l'affichage des messages, qui doit se faire de manière hiérarchique, avec pour chaque message l'ensemble de ses descendants, le nombre de niveaux n'étant pas limité. Comme souvent avec ce type de structure, une fonction récursive permet de résoudre le problème de manière élégante et concise.

La méthode `afficheMess()` est chargée d'afficher, pour un film, la liste des réponses à un message dont l'identifiant est passé en paramètre. Pour chacun de ces messages, on crée une ancre permettant de lui répondre, et, plus important, on appelle à nouveau (récursivement) la fonction `AfficheMess()` en lui passant l'identifiant du message courant pour afficher tous ses fils. La récursion s'arrête quand on ne trouve plus de fils.

Le code présente une subtilité pour la gestion de la vue. Ce que l'on doit afficher ici, c'est un arbre dont chaque nœud correspond à un message et constitue la racine du sous-arbre correspondant à l'ensemble de ses descendants. Pour l'assemblage final avec le moteur de templates, on doit associer un nom d'entité à chaque nœud. C'est le rôle de la variable `nom_groupe` ci-dessous qui identifie de manière unique le nœud correspondant à un message et à ses descendants par la chaîne `group`*`id`*, où *`id`* est l'identifiant du message. La fonction `affichemess()` renvoie la représentation HTML du nœud courant, ce qui correspond donc à une instanciation de bas vers le haut de l'ensemble des nœuds constituant l'arborescence.

```
private function afficheMess ($id_film, $id_pere)
{
  // Recherche des messages fils du père courant
  $requete = "SELECT * FROM Message "
  . "WHERE id_film='$id_film' AND id_pere='$id_pere'";

  // On crée une entité nom_groupe pour placer la présentation
  // des réponses au message id_pere
  $nom_groupe = "groupe" . $id_pere;
  $this->vue->setVar($nom_groupe, "");

  // Affiche des messages dans une liste, avec appels récursifs
  $resultat = $this->bd->execRequete ($requete);
  while ($message = $this->bd->objetSuivant ($resultat)) {
    // Appel récursif pour obtenir la mise en forme des
    // réponses
    $this->vue->reponses = $this->afficheMess ($id_film,
        $message->id);

    // On place les informations dans la vue
    $this->vue->texte_message = $message->texte;
    $this->vue->id_pere = $message->id;
    $this->vue->sujet = $message->sujet;
    // Attention à bien coder le texte placé dans une URL
    $this->vue->sujet_code = urlEncode($message->sujet);
    $this->vue->email_createur = $message->email_createur;
```

```
        // Décodage de la date Unix
        $idate = getDate($message->date_creation);
        // Mise en forme de la date décod
        $this->vue->date_message =
        $idate['mday'] . "/" . $idate['mon'] . "/" . $idate['year'
            ];
        $this->vue->heure_message = $idate['hours'] . " heures " .
            $idate['minutes'];

        if ($id_pere != 0)
        $this->vue->prefixe= "RE : ";
        else
        $this->vue->prefixe = "";
        // Création de la présentation du message courant, et
        // concaténation dans l'entité $nom_groupe
        $this->vue->append($nom_groupe, "message");
    }
    // On renvoie les messages du niveau courant, avec toutes
    // leurs réponses
    return $this->vue->getVar($nom_groupe);
}
```

Au moment de l'appel initial à cette fonction, on lui donne l'identifiant 0, ce qui revient à afficher au premier niveau tous les messages qui ne constituent pas des réponses. Ensuite, à chaque appel à `afficheMess()`, on ouvre une nouvelle balise <**ul**>. Ces balises seront imbriquées dans le document HTML produit, qui donnera donc bien une présentation hiérarchique des messages.

On peut remarquer ici le traitement des dates. Elles sont stockées dans la base sous la forme d'un « *timestamp* » Unix, qui se décode très facilement avec la fonction `getDate()`. Cette dernière renvoie un tableau (voir page 496) avec tous les éléments constituant la date et l'heure. Il reste à les mettre en forme selon le format souhaité.

```
// Décodage de la date Unix
$idate = getDate($message->date_creation);
// Mise en forme de la date décodée
$date_affiche =
   $date['mday'] . "/" . $idate['mon'] . "/" . $idate['year'];
$heure_affiche = $idate['hours'] . "heures " . $idate['minutes'];
```

7.4 RECOMMANDATIONS

Nous abordons maintenant le module de prédiction qui constitue l'aspect le plus original du site. D'une manière générale, la recommandation de certains produits en fonction des goûts supposés d'un visiteur intéresse de nombreux domaines, dont bien entendu le commerce électronique. Les résultats obtenus sont toutefois assez aléatoires, en partie parce que les informations utilisables sont souvent, en qualité comme en quantité, insuffisantes.

L'idée de base se décrit simplement. Au départ, on sait que telle personne a aimé tel film, ce qui constitue la base de données dont un tout petit échantillon est donné ci-dessous.

Personne	Films
Pierre	Van Gogh, Sacrifice, La guerre des étoiles
Anne	Van Gogh, La guerre des étoiles, Sacrifice
Jacques	Batman, La guerre des étoiles
Phileas	Batman, La guerre des étoiles, Rambo
Marie	Sacrifice, Le septième sceau

Maintenant, sachant que Claire aime *Van Gogh* (le film !), que peut-on en déduire sur les autres films qu'elle peut aimer ? Tous ceux qui aiment *Van Gogh* aiment aussi *Sacrifice*, donc ce dernier film est probablement un bon choix. On peut aller un peu plus loin et supposer que *Le septième sceau* devrait également être recommandé, puisque Pierre aime *Van Gogh* et *Sacrifice*, et que Marie aime *Sacrifice* et *Le septième sceau*.

La guerre des étoiles semble plaire à tout le monde ; c'est aussi une bonne recommandation, bien qu'on ne puisse pas en apprendre grand-chose sur les goûts spécifiques d'une personne. Enfin, il y a trop peu d'informations sur ceux qui aiment *Rambo* pour pouvoir en déduire des prédictions fiables.

Les techniques de *filtrage coopératif* (*collaborative filtering* en anglais) reposent sur des algorithmes qui tentent de prédire les goûts de personnes en fonctions de leurs votes (qui aime quoi), ainsi que de toutes les informations recueillies sur ces personnes (profession, âge, sexe, etc). Ces algorithmes étant potentiellement complexes, nous nous limiterons à une approche assez simple. Un des intérêts de ce type d'application est de faire appel intensivement à un aspect important de SQL, *les requêtes agrégat*, que nous n'avons pas vu jusqu'à présent.

7.4.1 Algorithmes de prédiction

Il existe essentiellement deux approches pour calculer des prédictions.

Approche par classification

On cherche à déterminer des groupes (ou classes) d'utilisateurs partageant les mêmes goûts, et à rattacher l'utilisateur pour lequel on souhaite réaliser des prédictions à l'un de ces groupes. De même, on regroupe les films en groupes/classes.

En réorganisant par exemple le tableau précédent avec les personnes en ligne, les films en colonnes, un 'O' dans les cellules quand la personne a aimé le film, on peut mettre en valeur trois groupes de films (disons, « Action », « Classique », et « Autres »), et deux groupes d'utilisateurs.

	Batman	Rambo	Van Gogh	Sacrifice	Le septième sceau	La guerre des étoiles
Pierre			O	O		O
Anne			O	O		O
Marie				O	O	
Jacques	O					O
Phileas	O	O	O			O

Il y a une assez forte similarité entre Jacques et Phileas, et toute personne qui se verrait affecter à leur groupe devrait se voir proposer un des films du groupe « Action ». C'est vrai aussi, quoique à un moindre degré, entre le premier groupe de personnes et les classiques. Les informations sont plus clairsemées, et le degré de confiance que l'on peut attendre d'une prédiction est donc plus faible. Enfin, en ce qui concerne *la guerre des étoiles*, il ne semble pas y avoir d'affinité particulière avec l'un ou l'autre des groupes d'utilisateurs.

Les algorithmes qui suivent cette approche emploient des calculs de distance ou de similarité assez complexes, et tiennent compte des attributs caractérisant chaque personne ou chaque film. De plus la détermination des groupes est coûteuse en temps d'exécution, même si, une fois les groupes déterminés, il est assez facile d'établir une prédiction.

Approche par corrélation

L'algorithme utilisé pour notre site effectue un calcul de corrélation entre l'internaute qui demande une prédiction et ceux qui ont déjà voté. La corrélation établit le degré de proximité entre deux internautes en se basant sur leurs votes, indépendamment de leurs attributs (âge, sexe, région, etc).

L'idée de départ est que pour prédire la note qu'un internaute a sur un film f, on peut en première approche effectuer la moyenne des notes des autres internautes sur f. En notant par $n_{a,f}$ la note de a sur f, on obtient :

$$n_{a,f} = \frac{1}{N}\sum_{i=1}^{N} n_{i,f} \tag{7.1}$$

Cette méthode assez grossière ne tient pas compte de deux facteurs. D'une part chaque internaute a une manière propre de noter les films, qui peut être plutôt positive ou plutôt négative. La note attribuée à un film par un internaute a ne devrait donc pas être considérée dans l'absolu, mais par rapport à la note moyenne m_a donnée par cet internaute. Si, par exemple, a donne en moyenne une note de 3,5, alors une note de 3 attribuée à un film exprime un jugement légèrement négatif, comparable à une note de 2,5 pour un internaute b dont la moyenne est de 3.

D'autre part il faut tenir compte de la corrélation $c_{a,b}$ (ou degré de proximité) entre deux internautes a et b pour estimer dans quelle mesure la note de b doit influer sur la prédiction concernant a. On obtient finalement une formule améliorée, qui effectue la moyenne des notes pondérée par le coefficient de corrélation :

$$n_{a,f} = m_a + \frac{1}{C}\sum_{i=1}^{N} c_{a,i}(n_{i,f} - m_i) \tag{7.2}$$

C est la somme des corrélations. Elle permet de normaliser le résultat. Il reste à déterminer comment calculer la corrélation entre deux utilisateurs *a* et *b*. Il y a plusieurs formules possibles. Celle que nous utilisons se base sur tous les films pour lesquels *a* et *b* ont tous les deux voté. Ils sont désignés par *j* dans la formule ci-dessous.

$$c_{a,b} = \frac{\sum_j (n_{a,j} - m_a)(n_{b,j} - m_b)}{\sqrt{\sum_j (n_{a,j} - m_a)^2 \sum_j (n_{b,j} - m_b)^2}} \tag{7.3}$$

On peut vérifier que si deux internautes ont voté exactement de la même manière, le coefficient de corrélation obtenu est égal à 1. Si, d'un autre côté, l'un des internautes vote de manière totalement « neutre », c'est-à-dire avec une note toujours égale à sa moyenne, on ne peut rien déduire et la corrélation est nulle. On peut noter également que la corrélation entre deux internautes est d'autant plus pertinente que le nombre de films qu'ils ont tous les deux notés est important.

Il y a beaucoup d'améliorations possibles – et souhaitables. Elles visent à résoudre (partiellement) les deux problèmes de base du filtrage coopératif : l'absence d'information et le fait que certains films sont peu représentatifs (comme *La guerre des étoiles* dans l'exemple ci-dessus).

7.4.2 Agrégation de données avec SQL

Toutes les requêtes vues jusqu'à présent pouvaient être interprétées comme une suite d'opérations effectuées ligne à ligne. De même leur résultat était toujours constitué de valeurs issues de lignes individuelles. Les fonctionnalités *d'agrégation* de SQL permettent d'exprimer des conditions sur des *groupes* de lignes, et de constituer le résultat en appliquant une fonction d'agrégation sur ce groupe. L'exemple le plus simple consiste à calculer le nombre de lignes dans une table.

```
SELECT COUNT(*)
FROM   Film
```

La fonction `COUNT()` compte le nombre de lignes. Ici, le groupe de lignes est constitué de la table elle-même. Il est bien entendu possible d'utiliser la clause `WHERE` pour sélectionner la partie de la table sur laquelle on applique la fonction d'agrégation.

```
SELECT COUNT(*)
FROM   Film
WHERE  genre = 'Western'
AND    annee > 1990
```

La fonction `COUNT()`, comme les autres fonctions d'agrégation, s'applique à tout ou partie des attributs de la table. On peut donc écrire également.

```
SELECT COUNT(id_realisateur)
FROM   Film
WHERE  genre = 'Western'
AND    annee > 1990
```

La différence entre les deux requêtes qui précèdent est subtile : **COUNT**(*expr*) compte en fait le nombre de lignes telles que la valeur de *expr* n'est pas à NULL. Si on utilise '*', comme dans le premier cas, on est sûr de compter toutes les lignes puisqu'il y a toujours au moins un attribut qui n'est pas à NULL dans une ligne (par exemple l'attribut titre est déclaré à NOT NULL : voir chapitre 4). En revanche la deuxième requête ne comptera pas les lignes où id_realisateur est à NULL.

Il n'est pas possible, sauf avec la clause GROUP BY qui est présentée plus loin, d'utiliser simultanément des noms d'attributs et des fonctions d'agrégation. La requête suivante est donc incorrecte :

```
SELECT titre , COUNT(id_realisateur)
FROM   Film
WHERE  genre = 'Western'
AND    annee > 1990
```

On demande en fait alors à MySQL deux choses contradictoires. D'une part il faut afficher tous les titres des westerns parus après 1990, d'autre part donner le nombre des réalisateurs de ces westerns. Il n'y a pas de représentation possible de cette information sous forme d'une table avec des lignes, des colonnes, et une seule valeur par cellule, et SQL, qui ne sait produire que des tables, rejette cette requête. La liste des fonctions d'agrégation est donnée dans la table 7.1.

Tableau 7.1 — Les fonctions d'agrégation de MySQL

Fonction	Description
COUNT(*expression*)	Compte le nombre de lignes.
AVG(*expression*)	Calcule la moyenne de *expression*.
MIN(*expression*)	Calcule la valeur minimale de *expression*.
MAX(*expression*)	Calcule la valeur maximale de *expression*.
SUM(*expression*)	Calcule la somme de *expression*.
STD(*expression*)	Calcule l'écart-type de *expression*.

Les requêtes dont nous avons besoin pour nos prédictions calculent des moyennes. La moyenne des notes pour l'internaute *fogg@foo.fr* est obtenue par :

```
SELECT AVG(note)
FROM   Notation
WHERE  email = 'fogg@foo.fr'
```

Symétriquement, la moyenne des notes pour un film –par exemple, *Vertigo*– est obtenue par :

```
SELECT AVG(note)
FROM   Notation
WHERE  titre = 'Vertigo'
```

La clause `GROUP BY`

Dans les requêtes précédentes, on applique la fonction d'agrégation à l'ensemble du résultat d'une requête (donc potentiellement à l'ensemble de la table elle-même). Une fonctionnalité complémentaire consiste à *partitionner* ce résultat en groupes, pour appliquer la ou les fonction(s) d'agrégat à chaque groupe. On construit les groupes en associant les lignes partageant la même valeur pour un ou plusieurs attributs.

La requête suivante affiche les internautes avec leur note moyenne :

```
SELECT   email, AVG(note)
FROM     Notation
GROUP BY email
```

On constitue ici un groupe pour chaque internaute (clause `GROUP BY email`). Puis on affiche ce groupe sous la forme d'une ligne, dans laquelle les attributs peuvent être de deux types :

1. ceux dont la valeur est constante pour l'ensemble du groupe, soit précisément les attributs du `GROUP BY`, ici l'attribut `email` ;
2. le résultat d'une fonction d'agrégation appliquée au groupe : ici `AVG(note)`.

Bien entendu, il est possible d'exprimer des ordres SQL comprenant une clause `WHERE` et d'appliquer un `GROUP BY` au résultat. D'autre part, il n'est pas nécessaire de faire figurer tous les attributs du `GROUP BY` dans la clause `SELECT`. Enfin, le `AS` permet de donner un nom aux attributs résultant d'une agrégation.

Voici un exemple assez complet donnant la liste des films avec le nom et le prénom de leur metteur en scène, et la moyenne des notes obtenues, la note minimale et la note maximale.

```
SELECT   f.titre, nom, prenom, AVG(note) AS moyenne,
         MIN(note) AS noteMin, MAX(note) AS noteMax
FROM     Film AS f, Notation AS n, Artiste AS a
WHERE    f.titre = n.titre
AND      f.id_realisateur = a.id
GROUP BY f.titre, nom, prenom
```

L'interprétation est la suivante : (1) on exécute d'abord la requête `SELECT ... FROM ... WHERE`, puis (2) on prend le résultat et on le partitionne, enfin (3) on calcule le résultat des fonctions pour chaque groupe.

7.4.3 Recommandations de films

Deux méthodes de recommandations sont proposées, toutes deux implantées dans le contrôleur `Recomm`. L'internaute peut choisir le nombre de films à afficher (10 par défaut) et appuyer sur un des boutons correspondant aux deux choix proposés (voir figure 7.5). L'action `proposer` associée au formulaire effectue quelques tests initiaux pour vérifier qu'un calcul de proposition est bien possible, et se contente d'appeler la fonction appropriée. L'affichage des films, standard, n'est pas montré dans le code ci-dessous. Il figure bien sûr dans les fichiers du site.

Webscope – Pratique de MySQL et PHP
Recommandation des films
À partir de cette page vous pouvez demander au serveur de vous recommander des films,
Deux méthodes de recommandations vous sont proposées.
1. La première affiche simplement la liste des films les mieux notés en moyenne.
2. La seconde recherche des films que vous n'avez pas noté (et donc peut-être pas vu) et essaye de prédire, par une note entre 1 et 5, le niveau de satisfaction que vous pouvez en attendre. Vous devez avoir noté au moins 10 films pour obtenir une prédiction.
Indiquez le nombre de films que vous souhaitez afficher puis appuyez sur le bouton correspondant à la méthode que vous avez choisie.
10 20 30
Films les plus appréciés
Prédiction
Menu
Accueil
Inscription
Authentification
Saisie
Modification
Recherche et notation
Recommandations
XML
Vous êtes Philippe Rigaux. Vous pouvez vous déconnecter à tout moment.

Figure 7.5 — Formulaire d'accès aux prédictions

```
function proposer()
{
  // D'abord on récupère les choix de l'utilisateur
  // Il faudrait vérifier qu'ils existent bien...
  $nb_films = $_POST['nb_films'];
  $liste_triee = isSet ($_POST['films_tries']);

  if ($nb_films <= 0 or $nb_films > 30) {
    $this->vue->contenu =
     "Ce script attend une variable nb_films comprise entre 1
         et 30\n";
  }
  else {
    // On vérifie que l'internaute a noté suffisamment de films
    $nb_notes = Util::NombreNotes($this->session->email, $this
        ->bd);

    // Message d'avertissement s'il n'y a pas assez de films
    if ($nb_notes < self::MIN_NOTES and !$liste_triee) {
      $this->vue->avertissement =
          "Vous n'avez pas noté assez de films ($nb_notes) "
          . "pour que nous puissions établir une prédiction !\n";
      $liste_triee = true;
    }
    else {
      $this->vue->avertissement = "";
    }

    $this->vue->nb_notes = $nb_notes;

    // Calcul de la liste des meilleurs films
```

```
    if ($liste_triee) {
     $films = $this->listeTriee ($nb_films);
     $this->vue->nb_films = $nb_films;
     $this->vue->setFile("contenu", "recomm_liste_triee.tpl");
    }
    else {
     // Calcul des prédictions
     $this->vue->ma_moyenne =
          Util::moyenneInternaute($this->session->email, $this
              ->bd);
     $films = $this->prediction ($this->session->email,
                  $nb_films, $this->bd);
     $this->vue->setFile("contenu", "recomm_liste_predite.tpl");
    }

    // Ensuite on affiche la liste des films -- Voir le code
  }
```

Les deux méthodes `listeTriee()` et `prediction()` correspondent respectivement aux deux types de propositions possibles. Elles sont toutes deux implantées comme des méthodes privées du contrôleur `Recomm` et données plus bas.

Liste des films les plus populaires

La méthode `listeTriee()` s'appuie sur les fonctionnalités d'agrégation de SQL pour construire la liste des films avec leur note moyenne. Le résultat est placé dans un tableau associatif, la clé étant l'identifiant du film et la valeur la note moyenne pour ce film.

Il ne reste plus qu'à trier sur la note, en ordre décroissant et à afficher les 10, 20 ou 30 premiers films de la liste triée. PHP fournit de nombreuses fonctions de tri de tableau (voir annexe C), dont `asort()` et `arsort()` pour trier sur les valeurs, et `ksort()` ou `krsort()` pour trier sur les clés un tableau associatif, respectivement en ordre croissant et en ordre décroissant. C'est `arsort()` qui nous intéresse ici.

Finalement, on affiche la liste des films (voir figure 7.6), en associant le titre à une ancre menant au contrôleur `Film` pour la présentation détaillée et le forum de discussion.

```
private function listeTriee ($nbFilms)
{
  $films = array();

  // Recherche des films et de leur note moyenne
  $classement = "SELECT id_film, AVG(note) AS note
  " . " FROM Notation GROUP BY id_film";
  $resultat = $this->bd->execRequete ($classement);
```

```
    // On crée un tableau associatif des films, avec leur note
    // moyenne
    $i=1;
    while ($film = $this->bd->objetSuivant ($resultat)) {
      $films[$film->id_film] = $film->note;
      if ($i++ >= $nbFilms) break;
    }

    // Tri du tableau par ordre décroissant sur note moyenne
    arsort ($films);

    return $films;
}
```

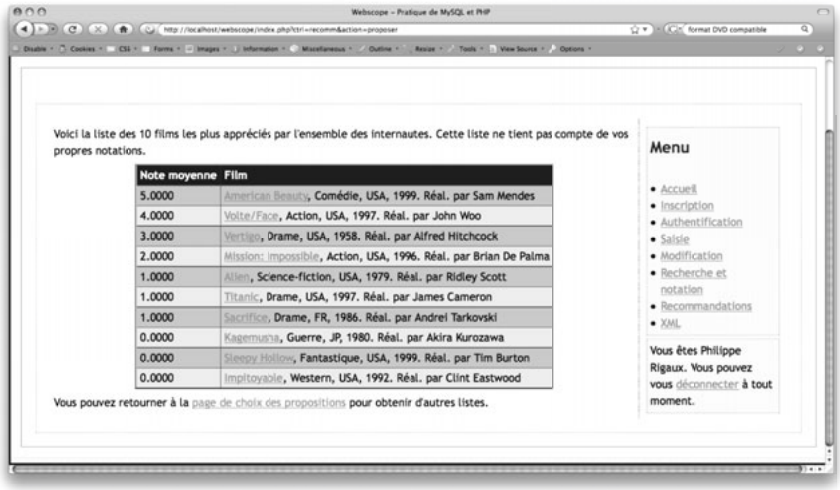

Webscope - Pratique de MySQL et PHP
Voici la liste des 10 films les plus appréciés par l'ensemble des internautes. Cette liste ne tient pas compte de vos propres notations.
Note moyenne Film
5.0000 American Beauty, Comédie, USA, 1999. Réal. par Sam Mendes
4.0000 Volte/Face, Action, USA, 1997. Réal. par John Woo
3.0000 Vertigo, Drame, USA, 1958. Réal. par Alfred Hitchcock
2.0000 Mission: impossible, Action, USA, 1996. Réal. par Brian De Palma
1.0000 Alien, Science-fiction, USA, 1979. Réal. par Ridley Scott
1.0000 Titanic, Drame, USA, 1997. Réal. par James Cameron
1.0000 Sacrifice, Drame, FR, 1986. Réal. par Andrei Tarkovski
0.0000 Kagemusha, Guerre, JP, 1980. Réal. par Akira Kurozawa
0.0000 Sleepy Hollow, Fantastique, USA, 1999. Réal. par Tim Burton
0.0000 Impitoyable, Western, USA, 1992. Réal. par Clint Eastwood
Vous pouvez retourner à la page de choix des propositions pour obtenir d'autres listes.
Menu
Accueil
Inscription
Authentification
Saisie
Modification
Recherche et notation
Recommandations
XML
Vous êtes Philippe Rigaux. Vous pouvez vous déconnecter à tout moment.

Figure 7.6 — Liste des films les plus appréciés

Calcul des prédictions

Le calcul des prédictions est nettement plus complexe que le tri des films sur leur note moyenne. La méthode `prediction()` fait elle-même appel à quelques autres fonctions implantées comme des méthodes statiques de la classe `Util`.

1. *MoyenneInternaute*(`email`) calcule la note moyenne de l'internaute identifié par `email`.
2. *ChercheNotation*(`email, id_film`), déjà rencontrée, recherche la notation d'un internaute sur un film.
3. *CalculCorrelation*(`email1, email2`), calcule le coefficient de corrélation entre deux internautes.
4. *CalculPrediction*(`email, id_film, tableauCorrelation`), prédit la note d'un film pour un internaute, étant donné le tableau des corrélations entre cet internaute et tous les autres.

Nous ne donnerons pas les deux premières fonctions ci-dessus qui se contentent d'exécuter une requête SQL (une agrégation pour `MoyenneInternaute()`) et

renvoient le résultat. Elles se trouvent dans *classes/Util.php*. Les deux autres fonctions se chargent de calculer respectivement les formules (7.3) et (7.2), page 306. Afin de calculer une prédiction, on procède en deux étapes :

1. d'abord on calcule le coefficient de corrélation entre l'internaute pour lequel on s'apprête à faire la prédiction et tous les autres : on stocke ces valeurs dans un tableau associatif `$tabCorr` indexé par le nom des internautes ;
2. puis on prend chaque film non noté par l'internaute, on applique la fonction calculant la prédiction, et on affiche le résultat.

Le calcul des prédictions pour un ensemble de films est implanté par la méthode privée `prediction()` du contrôleur `Recomm`, donnée ci-dessous.

```
private function prediction ($email, $nb_films)
{
  $films = array();

  // Calcul des corrélations avec les autres internautes
  $reqInternautes = "SELECT i.* FROM Internaute i, Notation n "
  . " WHERE n.email = i.email AND  i.email != '$email' "
  . " GROUP BY i.email HAVING COUNT(*) > 10";

  $listeInter  = $this->bd->execRequete ($reqInternautes);
  $tab_corr = array();
  while ($internaute =  $this->bd->objetSuivant ($listeInter))
    $tab_corr[$internaute->email] =
      Util::calculCorrelation ($email, $internaute->email,
          $this->bd);

  // Recherche des films, en ordre aléatoire
  $requete = "SELECT *  FROM Film ORDER BY RAND()";
  $resultat = $this->bd->execRequete ($requete);
  $i=1;
  while ($film =  $this->bd->objetSuivant ($resultat)) {
    // On vérifie que ce film n'est pas noté par l'internaute
    $notation = Util::chercheNotation ($email, $film->id, $this
        ->bd);

    if (!$notation)  {
      // Calcul de la prédiction pour ce film
      $prediction = Util::calculPrediction ($email, $film->id,
          $tab_corr, $this->bd);
      $prediction =  round ($prediction*100)/100;

      $films[$film->id] = $prediction;
      if ($i++ >= $nb_films) break;
    }
  }
  // On renvoie le tableau des prédictions
  return $films;
}
```

On constate qu'il faut manipuler beaucoup d'informations pour arriver au résultat, ce qui risque de soulever des problèmes de performance pour une base de données volumineuse. En particulier, le tableau des corrélations contient autant d'éléments qu'il y a d'internautes dans la base, et le passage de ce tableau en paramètre de la fonction `CalculPrediction()` peut être assez pénalisant. Un passage par référence éviterait cela.

Une fois qu'une application est stabilisée, et dans la mesure où elle est conçue de manière véritablement modulaire, il est relativement facile de remettre en cause quelques fonctions pour améliorer les performances. Ici une optimisation simple consisterait à ne pas recalculer systématiquement et en temps réel les corrélations entre internautes, mais à le faire périodiquement – par exemple une fois par jour – en stockant le résultat dans une table MySQL. Au lieu de passer le tableau `$tabCorr` en paramètre de la fonction `CalculPrediction()`, cette dernière pourrait alors chercher les corrélations dans la table.

La méthode statique `Util::calculCorrelation()` calcule et renvoie le coefficient de corrélation entre deux internautes, selon la formule (7.3). Elle effectue donc une boucle sur tous les films, prend ceux qui ont été notés par les deux internautes, et calcule les composants de la formule.

```
static function calculCorrelation ($email1, $email2, $bd)
  {
    $somme_numerateur = 0.;
    $somme_denom1 = 0.;
    $somme_denom2 = 0.;

    $moyenne1 = self::moyenneInternaute ($email1, $bd);
    $moyenne2 = self::moyenneInternaute ($email2, $bd);

    $requete = "SELECT * FROM Film";
    $listeFilms = $bd->execRequete ($requete);
    while ($film = $bd->objetSuivant ($listeFilms))
    {
      $notation1 = self::chercheNotation ($email1, $film->titre,
          $bd);
      $notation2 = self::chercheNotation ($email2, $film->titre,
          $bd);
      if ($notation1 and $notation2)
      {
        $somme_numerateur += ($notation1->note - $moyenne1)
        * ($notation2->note - $moyenne2);
        $somme_denom1 +=  pow($notation1->note - $moyenne1,2);
        $somme_denom2 +=  pow($notation2->note - $moyenne2,2);
      }
    }

    $somme_denominateur = sqrt ($somme_denom1 * $somme_denom2);
    if ($somme_denominateur != 0)
    $corr = abs($somme_numerateur)/$somme_denominateur;
    else
    $corr = 0;
```

```
    return $corr;
  }
```

Les fonctions `pow()`, `sqrt()` et `abs()` font partie du riche ensemble de fonctions de PHP (voir annexe C). Finalement la méthode statique `Util::calculPrediction()` applique la formule (7.2) pour calculer la prédiction sur un film comme la moyenne des notations des autres internautes sur ce film, pondérées par les coefficients de corrélation.

```
static function calculPrediction ($email, $id_film, $tab_corr,
    $bd)
{
  // Calcul de la moyenne des notes de l'internaute courant
  $ma_moyenne =  self::moyenneInternaute($email, $bd);

  // Boucle sur toutes les autres notations du même film
  $req_notations = "SELECT * FROM Notation WHERE id_film = '
      $id_film'";
  $liste_notations  = $bd->execRequete ($req_notations);

  // Application de la formule de corrélation
  $somme_corr = 0.;
  $somme_ponderee = 0.;
  while ($notation =  $bd->objetSuivant ($liste_notations)) {
    $moyenne =  self::moyenneInternaute($notation->email, $bd);
    $somme_corr += (float)  $tab_corr["$notation->email"];
    $somme_ponderee += (float) $tab_corr["$notation->email"] *
    ($notation->note - $moyenne);
  }
  if ($somme_corr != 0.)
    return  $ma_moyenne + ($somme_ponderee / $somme_corr);
  else
    return $ma_moyenne;
}
```

On peut remarquer dans cette fonction que les variables `$sommeCorr` et `$sommePonderee` sont explicitement manipulées comme des réels, avec la commande de conversion `(float)` placée devant une expression. Comme PHP est libre de convertir une variable en fonction du type qui lui semble le plus approprié, il vaut mieux prendre la précaution de donner explicitement ce type s'il est important.

8

XML

Ce chapitre propose une introduction à XML et présente quelques utilisations possibles de ce langage dans le cadre d'un site web basé sur MySQL et PHP. L'intérêt de XML dans un tel contexte consiste essentiellement à faciliter *l'échange* de données, aussi bien pour exporter des données de la base MySQL et les transmettre sur le réseau, que pour récupérer des données et les insérer dans la base. Nous verrons comment représenter une base de données relationnelle en XML et comment utiliser les fonctions PHP pour extraire des données de cette représentation.

Une autre possibilité d'utilisation de XML est la *publication* de données. Un des atouts de XML est de rendre la représentation de l'information indépendante des applications qui la manipulent. Dans notre cas, cela signifie qu'un document XML produit par un site web MySQL/PHP n'est pas réduit à être affiché dans un navigateur, à la différence des documents HTML construits jusqu'à présent. On peut, après une phase de transformation, proposer les informations de ce document au format HTML, mais aussi PDF pour une lecture ou impression de qualité, SVG pour des graphiques, ou enfin RSS. En d'autres termes on sépare le contenu de la présentation, ce qui rejoint en partie les objectifs des solutions de *templates* présentées dans le chapitre 5.

Notre présentation de XML est bien entendu illustrée dans le cadre du site WEBSCOPE, les fonctionnalités étant rassemblées dans le contrôleur `XML` dont la page d'accueil est reprise figure 8.1. On peut, à l'aide d'un formulaire semblable à celui utilisé pour la notation (voir page 295), rechercher des films et en obtenir une représentation sous forme de document XML, indépendante donc de MySQL, PHP ou HTML. On peut également appliquer une transformation XSLT à ce même document pour le mettre au format HTML (selon le même principe, on pourrait obtenir de nombreux autres formats). Enfin il est possible d'effectuer l'opération inverse, à savoir transmettre au serveur un fichier contenant un document XML

représentant des films, un script PHP se chargeant alors d'extraire les données du document pour les insérer dans la base MySQL.

Webscope – Pratique de MySQL et PHP
Le Webscope
Pratique de MySQL et PHP | Philippe Rigaux
Echange de données avec XML
Cette page vous permet de récupérer tout ou partie des films de la base de données au format XML et, inversement, de transmettre un document contenant un film pour qu'il soit inséré dans la base. Pour des raisons évidentes de sécurité, cette dernière fonctionnalité est inactivée sur les versions de démonstration du site.
Export de données XML
Le formulaire ci-dessous vous permet d'entrer des critères de recherche pour un ou plusieurs films. Vous pouvez également choisir le format de sortie : soit du XML simple, soit le résultat d'une transformation avec un programme XSLT pour produire du HTML.
Titre du film
Metteur en scène
Acteur
Genre
Année min.
Année max.
Format
Import d'un document XML
Vous pouvez transmettre un fichier contenant un document XML décrivant un film, et il sera inséré dans la base.
Fichier XML
Menu
accueil
inscription
authentification
saisie
modification
recherche et notation
recommandations
XML
Vous êtes Philippe Rigaux. Vous pouvez vous déconnecter à tout moment.

Figure 8.1 — Import et export de données XML dans le WEBSCOPE

8.1 INTRODUCTION À XML

XML (pour *eXtensible Markup Language*) est un langage (ou « méta-langage », langage pour définir d'autres langages) qui permet de structurer de l'information. Il utilise, comme HTML (et la comparaison s'arrête là), des *balises* pour « marquer » les différentes parties d'un contenu, ce qui permet aux applications qui utilisent celui-ci de s'y retrouver. Voici un exemple de contenu :

```
Le film Gladiator, produit aux États-Unis et réalisé par Ridley
                Scott, est paru en l'an 2000.
```

Il s'agit d'un texte simple et non structuré dont aucune application automatisée ne peut extraire avec pertinence les informations. Voici maintenant une représentation possible du même contenu en XML :

Exemple 8.1 *Gladiator.xml* : *Un contenu structuré avec XML*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- Document XML -->

<Film>
    <titre>Gladiator</titre>
    <annee>2000</annee>
```

```
    <code_pays>USA</code_pays>
    <genre>Drame</genre>
    <Realisateur
        nom="Scott"
        prenom="Ridley"
        annee_naissance="1937"/>
</Film>
```

Cette fois, des balises ouvrantes et fermantes séparent les différentes parties de ce contenu (titre, année de parution, pays producteur, etc.). Il devient alors (relativement) facile d'analyser et d'exploiter ce contenu, sous réserve bien entendu que la signification de ces balises soit connue.

8.1.1 Pourquoi XML ?

Où est la nouveauté ? Une tendance naturelle des programmes informatiques est de représenter l'information qu'ils manipulent selon un format qui leur est propre. Les informations sur le film *Gladiator* seront par exemple stockées selon un certain format dans MySQL, et dans un autre format si on les édite avec un traitement de texte comme Word. L'inconvénient est que l'on a souvent besoin de manipuler la même information avec des applications différentes, et que le fait que ces applications utilisent un format propriétaire empêche de passer de l'une à l'autre. Il n'y a pas moyen d'éditer avec Word un film stocké dans une base de données, et inversement, pas moyen d'interroger un document Word avec SQL. Le premier intérêt de XML est de favoriser l'interopérabilité des applications en permettant *l'échange* des informations.

Bien, mais on pourrait très bien répondre que le jour où on a besoin d'échanger des informations entre une application *A* et une application *B*, il suffit de réaliser un petit module d'export/import en choisissant la représentation la plus simple possible. Par exemple on peut exporter une table MySQL dans un fichier texte, avec une ligne de fichier par ligne de la table, et un séparateur de champ prédéfini comme le point-virgule ou la tabulation. À court terme, une telle solution paraît plus rapide et simple à mettre en œuvre qu'une mise en forme XML. À long terme cependant, la multiplication de ces formats d'échange est difficile à gérer et à faire évoluer. En choisissant dès le départ une représentation XML, on dispose d'un langage de structuration beaucoup plus puissant qu'un format à base de sauts de lignes et de séparateurs arbitraires, et on bénéficie d'un grand nombre d'outils d'analyse et de traitement qui vont considérablement simplifier la réalisation et la maintenance. Ces outils (éditeurs, analyseurs, interfaces de programmation) constituent le second avantage de XML : plus besoin d'inventer un format spécial, et de développer à partir de rien les fonctions qui vont avec.

En résumé, XML fournit des règles de représentation de l'information qui sont « neutres » au regard des différentes applications susceptibles de traiter cette information. Il facilite donc l'échange de données, la transformation de ces données, et en général l'exploitation dans des contextes divers d'un même document.

8.1.2 XML et HTML

XML et HTML ont un ancêtre commun, le langage SGML. Il est tout à fait impropre de présenter XML comme un « super » HTML. Voici deux particularités de HTML qui contrastent avec les caractéristiques de XML décrites ci-dessus :

- HTML est lié à une utilisation spécifique : la mise en forme, dans un navigateur, de documents transmis par des serveurs web ;
- le vocabulaire de balises de HTML est fixé et principalement constitué de directives de mises en forme du document (sauts de ligne, paragraphes, tableaux, etc.) ; il est donc tout à fait impropre à structurer le contenu d'un document quelconque.

Il serait plus juste de décrire HTML comme une spécialisation, un « dialecte » de XML, spécifiquement dédié à la visualisation de documents sur le Web. Même cette interprétation reste cependant partiellement inexacte : les règles syntaxiques de balisage dans HTML sont en effet beaucoup moins strictes que pour XML. En particulier, il n'y a pas systématiquement de balise fermante associée à une balise ouvrante, les attributs n'ont pas toujours de valeur, cette valeur n'est pas nécessairement encadrée par des apostrophes, etc. Ces différences sont regrettables car elle empêchent de traiter un document HTML avec les outils standard XML.

Une tendance récente est le développement de « dialectes » XML (autrement dit d'un vocabulaire de noms de balises et de règles de structuration de ces balises) adaptés à la représentation des données pour des domaines d'application particuliers. On peut citer le langage SMIL pour les données multimédia, le langage SVG pour les données graphiques, le langage WML pour les téléphones mobiles, le langage XSL-FO pour les documents imprimables, enfin XHTML, reformulation de HTML conforme à la syntaxe XML que nous utilisons depuis le début de ce livre. Tous ces dialectes, définis par le *World Wide Web Consortium*, sont conformes aux règles syntaxiques XML, présentées ci-dessous, et les documents sont donc manipulables avec les interfaces de programmation standard XML.

8.1.3 Syntaxe de XML

Les documents XML sont la plupart du temps créés, stockés et surtout échangés sous une forme *sérialisée* qui « marque » la structure par des balises mêlées au contenu textuel. Ce marquage obéit à des règles syntaxiques, la principale étant que le parenthésage défini par les balises doit être imbriqué : si une balise `<b>` est ouverte entre deux balises `<a>` et `</a>`, elle doit également être fermée par `</b>` entre ces deux balises. Cette contrainte introduit une hiérarchie entre les éléments définis par les balises. Dans l'exemple 8.1, page 318, l'élément `<titre>Gladiator</titre>` est un sous-élément de l'élément défini par la balise `<Film>`. Ce dernier, qui englobe tout le document (sauf la première ligne), est appelé *l'élément racine*.

On peut noter que le nom des balises, ainsi que l'ordre d'imbrication de ces balises, sont totalement libres : il n'existe pas en XML de règles prédéfinies. Cela permet à chacun de définir son propre langage pour décrire ses données.

Un document XML est une chaîne de caractères qui doit toujours débuter par une *déclaration XML* :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Cette première ligne indique que la chaîne contient des informations codées avec la version 1.0 de XML, et que le jeu de caractères utilisé est conforme à la norme ISO-8859-1 définie par l'Organisation Internationale de Standardisation (ISO) pour les langues latines d'Europe de l'Ouest. Cette norme est adaptée à l'usage du français puisqu'elle permet les lettres accentuées commme le « `é` ».

Éléments

Les éléments constituent les principaux composants d'un document XML. Chaque élément se compose d'une *balise ouvrante* <*nom*>, de son *contenu* et d'une *balise fermante* </*nom*>. Contrairement à HTML, les majuscules et minuscules sont différenciées dans les noms d'élément. Tout document XML comprend un et un seul *élément racine* qui définit le contenu même du document.

Un élément a un nom, mais il n'a pas de « valeur ». Plus subtilement, on parle de *contenu* d'un élément pour désigner la combinaison arbitrairement complexe de commentaires, d'autres éléments, de références à des entités et de données caractères qui peuvent se trouver entre la balise ouvrante et la balise fermante.

La présence des balises ouvrantes et fermantes est obligatoire pour chaque élément. En revanche, le contenu d'un élément peut être vide. Il existe alors une convention qui permet d'abréger la notation en utilisant une seule balise de la forme <`nom/`>. Dans notre document, l'élément `Realisateur` est vide, ce qui ne l'empêche pas d'avoir des attributs.

Les noms de balise utilisés dans un document XML sont libres et peuvent comprendre des lettres de l'alphabet, des chiffres, et les caractères « `-` » et « `_` ». Néanmoins il ne doivent pas contenir d'espaces ou commencer par un chiffre.

Attributs

Dans le document *Gladiator.xml* l'élément `Realisateur` a trois *attributs*. Les attributs d'un élément apparaissent toujours dans la balise ouvrante, sous la forme *`nom="valeur"`* ou *`nom='valeur'`*, où *`nom`* est le nom de l'attribut et *`valeur`* est sa valeur. Les noms d'attributs suivent les mêmes règles que les noms d'éléments. Si un élément a plusieurs attributs, ceux-ci sont séparés par des espaces.

Un élément ne peut pas avoir deux attributs avec le même nom. De plus, contrairement à HTML, il est bon de noter que

- un attribut doit *toujours* avoir une valeur ;
- la valeur doit être comprise entre des apostrophes simples ('10') ou doubles ("10") ; la valeur elle-même peut contenir des apostrophes simples si elle est encadrée par des apostrophes doubles, et réciproquement.

Instructions de traitement

Les *instructions de traitement* (en anglais *processing instructions*) sont conçues pour intégrer des instructions propres à un processeur particulier dans un document XML. Ainsi l'instruction suivante indique à un processeur XSLT l'adresse du fichier qui contient le programme (ici, *prog.xslt*) pour la transformation du document :

```
<?xml-stylesheet href="prog.xslt" type="text/xslt"?>
```

Le traitement d'une instruction dépend du cycle de vie du document XML et des applications qui le traitent. Par exemple l'instruction précédente peut être traitée par le serveur web qui publie le document, par le navigateur web qui l'affiche ou pas du tout dans le cas où il est chargé dans un éditeur de texte standard.

Sections CDATA

Il peut arriver que l'on souhaite placer dans un document du texte qui ne doit pas être analysé par le parseur. C'est le cas par exemple :

1. quand on veut inclure (à des fins de documentation par exemple) du code de programmation qui contient des caractères réservés dans XML : les '<' et '&' abondent dans du code C ou C++ ;
2. quand le document XML est consacré lui-même à XML, avec des exemples de balises que l'on souhaite reproduire littéralement.

Le document XML suivant n'est pas *bien formé* par rapport à la syntaxe XML :

Exemple 8.2 *ProgrErr.xml* : *Une ligne de code C dans un document XML*

```
<?xml version='1.0'?>
<PROGRAMME>
if ((i < 5) && (j > 6)) printf("error");
</PROGRAMME>
```

Une analyse syntaxique du fichier *ProgrErr.xml* détecterait plusieurs erreurs liées à la présence des caractères '<', '>' et '&' dans le contenu de l'élément <PROGRAMME>.

Les *sections littérales* CDATA permettent d'éviter ce problème en définissant des zones non analysées par le parseur XML. Une section CDATA est une portion de texte entourée par les balises <![CDATA[et]]>. Ainsi, il est possible d'inclure cette ligne de code dans une section CDATA pour éviter une erreur syntaxique provoquée par les caractères réservés :

Exemple 8.3 *Progr1.xml* : *Une ligne de C dans une section CDATA*

```
<?xml version='1.0'?>
<PROGRAMME>
<![CDATA[if ((i < 5) && (j > 6)) printf("error");]]>
</PROGRAMME>
```

8.2 EXPORT DE DONNÉES XML

Passons maintenant à la pratique en commençant par la mise en forme d'une base de données relationelle en XML. Commençons par discuter des principes avant de montrer comment réaliser un module PHP qui permet « d'exporter » tout ou partie de notre base de films.

8.2.1 Comment passer d'une base MySQL à XML

Les règles de transformation d'une base relationnelle en XML ne vont pas forcément de soi. Il y a plusieurs choix à faire, dont certains sont naturels et d'autres plus ou moins arbitraires.

Éléments ou attributs ?

La première solution, immédiate, consiste à conserver la structure « plate » de la base relationnelle, et à transcrire chaque table par un élément ayant le nom de la table ou un dérivé (par exemple <Films>). Cet élément contient lui-même un élément pour chaque ligne, ayant pour nom un autre dérivé du nom de la table (par exemple « Film », sans « s »), enfin chaque attribut de la table est représenté par un élément, constituant ainsi un troisième niveau dans l'arbre.

Il existe (au moins) une autre possibilité. Au lieu de représenter les attributs de la table par des éléments, on peut les représenter par des attributs XML de l'élément représentant la ligne. Voici ce que cela donnerait pour la table *Film*, avec deux films.

Exemple 8.4 *FilmAttrs.xml* : *Représentation de Film avec des attributs*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Films>
        <Film
           titre="Sleepy Hollow"
           annee="1999"
           code_pays="USA"
           genre="Fantastique"
           id_realisateur="13"/>
        <Film
           titre="Eyes Wide Shut"
           annee="1999"
           code_pays="USA"
           genre="Thriller"
           id_realisateur="101"/>
</Films>
```

Cette méthode présente quelques avantages. Tout d'abord elle est assez proche, conceptuellement, de la représentation relationnelle. Chaque ligne d'une table devient un élément XML, chaque attribut de la ligne devient un attribut XML de l'élément. La structure est plus fidèlement retranscrite, et notamment le fait

qu'une ligne d'une table forme un tout, manipulé solidairement par les langages. En SQL par exemple, on n'accède jamais à un attribut sans être d'abord passé par la ligne de la table.

Techniquement, l'absence d'ordre (significatif) sur les attributs XML correspond à l'absence d'ordre significatif sur les colonnes d'une table. Du point de vue du typage, l'utilisation des attributs permet également d'être plus précis et plus proche de la représentation relationnelle :

- on ne peut pas avoir deux fois le même attribut pour un élément, de même qu'on ne peut pas avoir deux colonnes avec le même nom dans une table (ce n'est pas le cas si on représente les colonnes par des éléments XML) ;
- on peut indiquer, dans une DTD, la liste des valeurs que peut prendre un attribut, ce qui renforce un peu les contrôles sur le document.

Enfin, l'utilisation des attributs aboutit à un document moins volumineux. Comme pour beaucoup d'autres problèmes sans solution tranchée, le choix dépend en fait beaucoup de l'application et de l'utilisation qui est faite des informations.

Représentation des associations entre tables

Passons maintenant à la représentation XML des liens entre les tables. En relationnel, les liens sont définis par une correspondance entre la clé primaire dans une table, et une clé étrangère dans une autre table. En d'autres termes, la condition nécessaire et suffisante pour qu'il soit possible de reconstituer l'information est l'existence d'un critère de rapprochement. Il est tout à fait possible d'appliquer le même principe en XML. Voici par exemple un document où figurent des éléments de nom `Film` et de nom `Artiste`. Ces éléments sont indépendants les uns des autres (ici cela signifie que des informations apparentées ne sont pas liées dans la structure arborescente du document), mais on a conservé le critère de rapprochement.

Exemple 8.5 *FilmArtiste.xml* : *Des films et des artistes*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Films>
   <!-- Les films -->
        <Film
           titre="Eyes Wide Shut"
           annee="1999"
           code_pays="USA"
           genre="Thriller"
           id_realisateur="101"/>
        <Film
           titre="Sleepy Hollow"
           annee="1999"
           code_pays="USA"
           genre="Fantastique"
           id_realisateur="13"/>
```

```
   <!-- Les artistes -->
        <Artiste
           id="101"
           nom="Kubrick"
           prenom="Stanley"
           annee_naissance="1928"/>
        <Artiste
           id="13"
           nom="Burton"
           prenom="Tim"
           annee_naissance="1958"/>
</Films>
```

Maintenant, comme dans le cas du relationnel, il est possible de déterminer *par calcul*, la correspondance entre un film et son metteur en scène en se servant de l'identifiant de l'artiste présent dans l'élément <Film>. Cette représentation n'est cependant pas naturelle en XML et mène à quelques difficultés. Elle n'est pas naturelle parce que le metteur en scène fait partie de la description d'un film, et qu'il est inutile de le représenter séparément. Elle présente des difficultés parce que l'exploitation du document pour reconstituer toute l'information va être compliquée.

La bonne représentation dans ce cas consiste à représenter les attributs d'un film avec des éléments, et à *imbriquer* un élément supplémentaire de type `Artiste`. On peut du même coup s'épargner la peine de conserver l'identifiant de l'artiste puisque la correspondance est maintenant représentée par la structure, pas par un lien de navigation basé sur des valeurs communes. Voici ce que cela donne.

Exemple 8.6 *FilmImbrique.xml* : *Représentation avec imbrication*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Films>
  <Film>
    <titre>Sleepy Hollow</titre>
    <annee>1999</annee>
    <code_pays>USA</code_pays>
    <genre>Fantastique</genre>
    <Realisateur
           nom="Burton"
           prenom="Tim"
           annee_naissance="1958"/>
  </Film>
  <Film>
    <titre>Eyes Wide Shut</titre>
    <annee>1999</annee>
    <code_pays>USA</code_pays>
    <genre>Thriller</genre>
    <Realisateur
           nom="Kubrick"
```

```
            prenom="Stanley"
            annee_naissance="1928"/>
  </Film>
</Films>
```

Cette représentation est bien meilleure. Il est maintenant possible, pour un élément `Film`, d'accéder directement au metteur en scène, au prix d'une duplication des informations sur ce dernier, autant de fois qu'il y a de films. On a perdu la symétrie du schéma relationnel : le chemin d'accès privilégié à l'information est le film. Si on voulait chercher dans le document tous les films réalisés par un metteur en scène, on se trouverait face à une recherche un peu plus compliquée à effectuer.

Dans ce cas, une solution plus logique consiste sans doute à placer le metteur en scène comme élément de plus haut niveau, et à imbriquer dans cet élément tous les films qu'il a réalisés. Voilà ce que cela donne, avec Clint Eastwood :

Exemple 8.7 *ArtisteFilm.xml* : *Changement de l'ordre d'imbrication*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Films>
    <Realisateur>
            <nom>Eastwood</nom>
            <prenom>Clint</prenom>
            <annee_naissance>1930</annee_naissance>
         <Film
            titre="Les pleins pouvoirs"
            annee="1997"
            code_pays="USA"
            genre="Policier"/>
         <Film
            titre="Impitoyable"
            annee="1992"
            code_pays="USA"
            genre="Western"/>
    </Realisateur>
</Films>
```

Le progrès le plus notable ici est qu'on évite toute duplication. C'est possible parce que l'association est de type *un à plusieurs* (voir chapitre 4).

En revanche, dans le cas d'associations *plusieurs à plusieurs*, l'imbrication ne va pas de soi. Prenons par exemple l'association entre les films et les acteurs. Pour un film il peut y avoir plusieurs acteurs et réciproquement. Dans le schéma relationnel on a créé une table intermédiaire *Role* pour représenter cette association.

Il n'est pas évident de choisir l'ordre d'imbrication des éléments. Tout dépend de l'ordre de navigation employé dans l'application. Si on suppose par exemple que les accès se feront par les films, on peut choisir l'imbrication représentée dans l'exemple suivant :

Exemple 8.8 *FilmActeur.xml* : *Les films, et les acteurs imbriqués*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Films>
  <Film>
    <titre>Piège de cristal</titre>
    <annee>1988</annee>
    <code_pays>USA</code_pays>
    <genre>Action</genre>

    <Acteur>
            <prenom>Bruce</prenom>
            <nom>Willis</nom>
            <annee_naissance>1955</annee_naissance>
            <nom_role>McClane</nom_role>
    </Acteur>
  </Film>

  <Film>
    <titre>Pulp fiction</titre>
    <annee>1994</annee>
    <code_pays>USA</code_pays>
    <genre>Action</genre>

    <Acteur>
            <prenom>John</prenom>
            <nom>Travolta</nom>
            <annee_naissance>1954</annee_naissance>
            <nom_role>Vincent Vega</nom_role>
    </Acteur>
    <Acteur>
            <prenom>Bruce</prenom>
            <nom>Willis</nom>
            <annee_naissance>1955</annee_naissance>
            <nom_role>Butch Coolidge</nom_role>
    </Acteur>
  </Film>
</Films>
```

Il est très facile, à partir d'un film, d'accéder aux acteurs du film. En revanche, si on cherche, pour un acteur, tous les films qu'il a joués, c'est plus difficile. En introduisant une hiérarchie Film/Acteur, on a donc privilégié un chemin d'accès aux données. La représentation inverse est également possible :

Exemple 8.9 *ActeurFilm.xml* : *les acteurs, et les films imbriqués*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Acteurs>
  <Acteur>
   <prenom>Bruce</prenom>
```

```
    <nom>Willis</nom>
    <annee_naissance>1955</annee_naissance>

    <Film>
      <titre>Piège de cristal</titre>
      <annee>1988</annee>
      <code_pays>USA</code_pays>
      <genre>Action</genre>
      <nom_role>McClane</nom_role>
     </Film>

    <Film>
      <titre>Pulp fiction</titre>
      <annee>1994</annee>
      <code_pays>USA</code_pays>
      <genre>Action</genre>
      <nom_role>Butch Coolidge</nom_role>
    </Film>
  </Acteur>
</Acteurs>
```

Cette fois, en supposant que le point d'accès est toujours un acteur, on a toutes les informations relatives à cet acteur dans le même sous-arbre, ce qui va permettre d'y accéder efficacement et simplement. On voit en revanche que si on souhaite prendre comme point d'accès un film, les informations utiles sont réparties un peu partout dans l'arbre, et que leur reconstitution sera plus difficile.

La base de données que nous utilisons dans nos exemples est très simple. Il est clair que pour des bases réalistes présentant quelques dizaines de tables, la conception d'un schéma XML d'exportation doit faire des compromis entre l'imbrication des données et la conservation des correspondances clé primaire/clé étrangère sous forme de lien de navigation dans le document XML. Tout dépend alors des besoins de l'application, de la partie de la base qu'il faut exporter, et des chemins d'accès privilégiés aux informations qui seront utilisés dans l'exploitation du document.

8.2.2 Application avec PHP

La transformation d'une table MySQL en document XML est extrêmement simple puisqu'il suffit de créer une chaîne de caractères au format approprié. Une approche directe mais fastidieuse consiste à agir au cas par cas en engendrant « à la main » les balises ouvrante et fermante et leur contenu. Comme toujours il faut essayer d'être le plus générique possible : la fonction présentée ci-dessous prend un tableau associatif contenant une liste (*nom*, *valeur*) et crée une chaîne XML. Cette chaîne est un élément dont le nom est passé en paramètre (si la chaîne vide est passée pour le nom, seul le contenu de l'élément, sans les balises ouvrante et fermante, est renvoyé).

REMARQUE – Il faut, comme en HTML, être attentif à éviter d'introduire des caractères réservés comme <, >, ', " ou & dans les contenus XML. Le traitement par la fonction `htmlSpecialChars()` qui remplace ces caractères en appels d'entités convient parfaitement.

L'élément créé est précédé de tabulations afin de faciliter une mise en forme claire du document final, comme nous le montrerons plus loin. Enfin la représentation du tableau peut, au choix, reposer sur des attributs ou des éléments. Voici le code de la fonction, qui est techniquement une méthode statique de la classe utilitaire `EchangeXML`.

```
static function tableauVersXML ($tableau=array(),
  $nom_element="ligne", $nb_tab=0, $format=self::ELEMENTS)
{
  // Création d'une chaîne avec le nombre $nb_tab de
  // tabulations
  $tabs="";
  for ($i=0; $i < $nb_tab ; $i++) $tabs .= "\t";

  $chaine_XML = $attrs = "";

  // Mise en forme en fonction du format demandé
  if ($format == self::ELEMENTS)  {
    // On crée simplement un élément XML pour chaque attribut
    // de la table, et on concatène les éléments
    foreach ($tableau as $nom => $val) {
      // On retire les retours à la ligne du résumé
      if ($nom == "resume") $val = str_replace("\n", "\n\t",
          $val);

      // On place l'identifiant comme un attribut
      if ($nom=="id") $attrs .=" $nom='$val' ";

      // Pour tous les autres on crée un élément
      if ($nom != "id_realisateur" and $nom!="id" and !empty(
          $val)) {
        $chaine_XML = $chaine_XML . $tabs
        . "    <$nom>" . htmlSpecialChars($val) . "</$nom>\n";
      }
    }
    // La chaîne obtenue est le contenu de l'élément
       $nom_element
    if (!empty($nom_element)) {
      $chaine_XML = "$tabs<$nom_element $attrs>\
          n$chaine_XML$tabs</$nom_element>\n";
    }
  }
  else  {
    // On crée un seul élément avec des attributs XML
    if (is_array($tableau)) {
      foreach($tableau as $nom => $val) {
        $chaine_XML .=  " $nom=\"" .htmlSpecialChars($val) . "
            \" ";
```

```
        }
      }
      $chaine_XML = "$tabs<$nom_element $chaine_XML/>\n";
    }
    return $chaine_XML;
  }
```

Les commentaires indiquent les étapes de cette conversion vers XML, qui ne présente aucune difficulté conceptuelle. Maintenant il devient très facile de transformer une base en document XML. Notre outil d'export (voir la copie d'écran page 318) offre un formulaire permettant à l'utilisateur de saisir des critères de recherche pour des films de la base. À partir de ces critères une requête est créée (on réemploie bien entendu la fonction `Util::creerRequetes()` déjà utilisée pour rechercher les films à noter), exécutée, et chaque film est mis sous la forme d'un élément XML auquel on ajoute le metteur en scène et les acteurs. Voici le script complet de l'action `export` dans le contrôleur `XML` :

```
 function export ()
{
  // Pour créer un fichier par film
  $multi_files=false;

  // Création de la requête SQL en fonction des critères
  $requete = Util::creerRequetes ($_POST, $this->bd);
  $resultat = $this->bd->execRequete ($requete);

  // On parcourt les films et on les transforme en XML
  $document = "";
  $nbFilms = 0;
  while ($film = $this->bd->ligneSuivante ($resultat)) {
    // Mise en forme du film
    $film_XML = EchangeXML::tableauVersXML ($film, "");

    // Mise en forme du metteur en scène
    $mes = Util::chercheArtisteAvecID($film['id_realisateur'],
    $this->bd, FORMAT_TABLEAU);

    $film_XML .= EchangeXML::tableauVersXML($mes, "realisateur",
    1, EchangeXML::ELEMENTS);

    // Ajout des acteurs et de leur rôle
    $req_acteurs = "SELECT id, prenom, nom, annee_naissance,
        nom_role "
    . "FROM Artiste A, Role R "
    . "WHERE A.id = R.id_acteur AND R.id_film ='{$film['id']}'";
    $res_acteurs = $this->bd->execRequete($req_acteurs);
    while ($role = $this->bd->ligneSuivante($res_acteurs)) {
      $film_XML .=  EchangeXML::tableauVersXML($role, "Acteur",
      1, EchangeXML::ELEMENTS);
```

```
        }
        // On place le contenu dans la balise <Film>
        $document .= "  <Film>\n" . $film_XML . "\n  </Film>\n";
        $nbFilms++;
    }

    // On envoie l'en-tête HTTP, et le prologue du document XML
    Header ("Content-type: text/xml");
    echo "<?xml version=\"1.0\" encoding=\"iso-8859-1\"?>\n\n";

    // Mise en forme selon le choix de l'utilisateur
    if ($_POST['format'] == "XML") {
        // On sort le XML brut
        echo "<Films>\n$document</Films>\n";;
    }
    else {
        // On applique une transformation XSLT. Il suffit d'ajouter
        // une instruction pour que le navigateur en tienne compte
        // et applique la transformation Film.xsl

        echo "<?xml-stylesheet href='./xsl/Film.xsl' type='text/xsl
            '?>\n"
                . "<Films>\n$document</Films>\n";;
    }
}
```

Quand le format choisi est XML, le document renvoyé est déclaré de type MIME `text/xml` pour qu'il soit affiché sous une forme présentable dans le navigateur. En jouant sur le type MIME on pourrait également forcer le téléchargement du document sur la machine du client `application/force-download`). Si on choisit le format HTML, le même document est transmis, mais avec une instruction de traitement qui demande au navigateur d'appliquer une transformation XSLT. Nous y revenons en fin de chapitre, page 348.

Vous pouvez directement utiliser ce script sur notre site pour récupérer un ou plusieurs films en XML. Voici par exemple le résultat obtenu pour *Kill Bill*.

Exemple 8.10 *KillBill.xml* : *Exemple de document produit par le script précédent*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Films>
 <Film>
  <titre>Kill Bill</titre>
  <annee>2003</annee>
  <code_pays>USA</code_pays>
  <genre>Drame</genre>
  <resume>Au cours d'une cérémonie de mariage en plein désert, un commando
  fait irruption dans la chapelle et tire sur les convives. Laissée pour
  morte, la Mariée enceinte retrouve ses esprits après un coma de quatre ans.
  Celle qui a auparavant exercé les fonctions de tueuse à gages au sein du
  Détachement International des Vipères Assassines n'a alors plus qu'une
```

```
seule idée en tête : venger la mort de ses proches en éliminant tous les
membres de l'organisation criminelle, dont leur chef Bill qu'elle se
réserve pour la fin.</resume>
      <Realisateur
          nom="Tarantino"
          prenom="Quentin"
          annee_naissance="1963" />
      <Acteur
          prenom="Uma"
          nom="Thurman"
          annee_naissance="1970"
          nom_role="La mariée, alias &quot;Black Mamba&quot;" />
      <Acteur
          prenom="Lucy"
          nom="Liu"
          annee_naissance="1968"
          nom_role="O-Ren Ishii" />
      <Acteur
          prenom="David"
          nom="Carradine"
          annee_naissance="1936"
          nom_role="Bill" />
      <Acteur
          prenom="Michael"
          nom="Madsen"
          annee_naissance="1958"
          nom_role="Budd / Sidewinder" />
      <Acteur
          prenom="Daryl"
          nom="Hannah"
          annee_naissance="1960"
          nom_role="Elle Driver" />

 </Film>
</Films>
```

8.3 IMPORT DE DONNÉES XML DANS MySQL

L'opération inverse, l'import d'un document XML dans une base de données MySQL, est un peu plus difficile. Au lieu de s'appuyer sur SQL pour récupérer les données dans la base, il faut utiliser un *parseur* de documents XML qui va analyser la structure du document et permettre d'accéder à ses différents composants.

Les parseurs XML s'appuient sur deux modèles possibles de traitement d'un document, connus respectivement sous les acronymes SAX (*Simple API for XML*) et DOM (*Document Object Model*). Le modèle de traitement de SAX consiste à parcourir le document linéairement, et à déclencher des fonctions à chaque fois qu'une des catégories syntaxiques (balises ouvrantes, fermantes, texte, instructions de traitement, etc.) constituant un document XML est rencontrée.

Le modèle DOM s'appuie sur une représentation arborescente. Chaque nœud de l'arbre est un objet, doté de méthodes propres au type du nœud, et de pointeurs vers le ou les sous-arbres, le père du nœud, les attributs, etc. On utilise plutôt DOM pour les applications qui doivent disposer en mémoire de l'ensemble de la représentation d'un document, comme par exemple un éditeur XML, un processeur de transformations XSLT, le langage de requête XQuery, etc. DOM est connu pour être gourmand en mémoire et parfois lent, et il est préférable d'éviter d'y recourir quand c'est possible. Pour en savoir plus sur DOM, vous pouvez consulter la recommandation du W3C, disponible sur le site *http://www.w3c.org/dom*.

PHP propose de plus une interface de manipulation de données XML, dite *SimpleXML*, qui fournit quelques fonctions très simples pour accéder au contenu d'un document. L'arborescence XML est représentée par *SimpleXML* comme une imbrication de tableaux PHP, accompagnée de quelques fonctions pour rechercher des éléments ou des attributs. *SimpleXML* peut être vue comme une version très basique de DOM (PHP fournit d'ailleurs une conversion depuis un objet DOM vers un objet SimpleXML).

La présentation qui suit montre successivement comment traiter un document XML avec SimpleXML, puis SAX, à chaque fois dans l'optique d'extraire des données du document pour les insérer dans MySQL. L'annexe C, page 500 récapitule les fonctions utilisées.

8.3.1 SimpleXML

Le fonctionnement de SimpleXML est comparable à celui d'une fonction comme `mysql_fetch_object()` : on crée un objet PHP (instance de la classe prédéfinie `SimpleXMLElement`) contenant une représentation de la source de données externe. Dans le cas de `mysql_fetch_object()`, la source de données est une ligne d'une table relationnelle, ce qui se représente simplement par une liste d'attributs de l'objet. Dans le cas de SimpleXML, l'objet créé a une structure nettement plus complexe. Elle s'appuie sur une représentation arborescente constituée d'une hiérarchie d'éléments dotée d'un unique élément racine. Voici les règles de construction qui aident à comprendre la représentation :

- l'objet instancié par SimpleXML correspond à l'élément racine du document ;
- les attributs publics de cet objet sont les éléments-fils de l'élément racine ; eux-mêmes sont des objets PHP de la classe `SimpleXMLElement`, ou un tableau d'objets s'il y a plusieurs occurrences d'un élément de même nom ;
- les attributs sont stockés dans une propriété privée de l'objet, accessible grâce à la méthode `attributes()`.

Ces règles s'appliquent, récursivement, aux éléments-fils de l'élément racine et à tous ses descendants. Prenons le cas du document XML *KillBill.xml*, page 331. Après analyse par SimpleXML, on obtiendra un objet référençant l'élément racine (correspondant à la balise `<Films>` du document). Les propriétés de cet objet sont les fils de l'élément racine, soit `titre`, `annee`, `code_pays`, `genre`, `resume`, `id_realisateur`, `Realisateur` et `Acteur`. Ce dernier est

un tableau indicé d'objets puisqu'on trouve plusieurs occurrences de l'élément <Acteur>. Voici un exemple de code SimpleXML extrayant des informations de cette structure.

Exemple 8.11 *exemples/ExSimpleXML.php* : *Application de l'API SimpleXML*

```
<?xml version="1.0" encoding="iso-8959-1"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
<head>
<title>Création d'un formulaire</title>
<link rel='stylesheet' href="films.css" type="text/css"/>
</head>
<body>
<?php
// Application des fonctions SimpleXML

// Analyse du document KillBill.xml
$doc = SimpleXML_load_file("KillBill.xml");

// Accès à un élément: le titre
echo "Film : " .$doc->Film[0]->titre . "<br>";

// Accès aux attributs de l'élément <Realisateur>
$attr_real = $doc->Film->Realisateur->attributes();
echo "Réalisé par " . $attr_real['prenom'] . " " . $attr_real['
    nom'];

// Affichage de la liste des acteurs
echo "<p>Avec: <ol>\n";

// Boucle sur l'ensemble des acteurs
foreach ($doc->Film->Acteur as $acteur) {
  // On prend les attributs du noeud courant
  $attributs = $acteur->attributes();
  // On les affiche
  echo "<li>" . $attributs['prenom'] . " " . $attributs['nom']
    . " dans le rôle de "
    . utf8_decode($attributs['nom_role']) . "</li>";
}
echo "</ol>";
?>
</body>
</html>
```

On accède donc aux éléments du document XML par une simple navigation dans une hiérarchie d'objets PHP. La variable `$doc` représentant l'élément racine, on obtient le titre avec `$doc->Film->titre`, le réalisateur avec

`$doc->Film->Realisateur`, et les acteurs avec `$doc->Film->Acteur`. Pour ce dernier on effectue une boucle avec `for each`. La méthode `attributes()` renvoie un tableau associatif contenant les attributs d'un élément. Noter que *SimpleXML* traite les chaînes de caractères en les codant en UTF-8, ce qui oblige parfois à les transférer en ISO-8859-1 quand elles contiennent des caractères accentués.

En résumé, SimpleXML offre une interface simple et pratique, quoique limitée, à un document XML de petite taille. Pour une exploitation généralisable à de gros documents, il reste préférable de recourir à l'API SAX, présentée ci-dessous.

8.3.2 L'API SAX

Les fonctions proposées par PHP s'appuient sur le parseur *expat* développé par James Clark (voir le site *http://www.jclark.com*). Elles sont disponibles systématiquement dans le cas d'une configuration de PHP avec Apache, ou peuvent être incluses avec l'option `--with-xml` sinon. L'analyse d'un document XML s'effectue en trois phases :

1. on initialise un parseur avec la fonction `xml_parser_create()` ;
2. on indique au parseur les fonctions à associer aux différents types de marquage rencontrés dans le document ;
3. enfin on lance l'analyse avec la fonction `xml_parse()`.

La seconde étape doit être adaptée à chaque type de document traité. Pour être concret voici un premier exemple d'un module d'analyse de documents XML, s'appliquant à n'importe quel document.

Exemple 8.12 *exemples/SAX.php* : *Un exemple simple de traitement d'un document XML*

```
<?php

  /** Analyse avec le parseur SAX d'un document XML.
  * La fonction renvoie un tableau associatif contenant toutes
      les
  * informations trouvées
  */

$tab_elements = array ();
$element_courant = "";

/**
 * Fonction déclenchée sur une balise ouvrante
  */
function debutElement ($parser, $nom, $attrs)
{
  global $element_courant;
  $element_courant = $nom;
  $tab_elements[$element_courant] = "";

  echo "balise ouvrante de $nom\n";
}
```

```
/**
 * Fonction déclenchée sur une balise fermante
 */

function finElement ($parser, $nom)
{
  echo "balise fermante de $nom\n";
}

/**
 * Fonction déclenchée sur du texte
 */

function donneesCaracteres ($parser, $chaine)
{
  global $tab_elements, $element_courant;

  if (trim($chaine) != "")
    $tab_elements[$element_courant] = $chaine;
}

/**
 * Cette fonction prend un nom de fichier contenant
 * un document XML et en extrait des informations
 */

function parseFilm ($nom_fichier)
{
  global $tab_elements;

  if (! ($f = fopen($nom_fichier, "r"))) {
    echo "Impossible d'ouvrir le  fichier $nom_fichier!!\n";
    return 0;
  }

  $parseur = xml_parser_create ();

  // Déclencheurs pour les éléments
  xml_set_element_handler ($parseur, "debutElement", "finElement"
      );

  // Déclencheurs pour les noeuds texte
  xml_set_character_data_handler ($parseur, "donneesCaracteres");

  // Lecture du document
  $document = fread ($f, 100000);

  xml_parse ($parseur, $document, feof($f));

  xml_parser_free ($parseur);

  return $tab_elements;
}
```

La fonction la plus importante est `xml_parse()`[1]. Elle prend en argument le parseur, le nom d'un fichier contenant un document XML à analyser, et un Booléen indiquant si le document est passé en totalité ou par fragments. L'analyse se résume à afficher au fur et à mesure les balises ouvrante et fermante rencontrées, et à stocker dans un tableau associatif le contenu des éléments.

Après la création du parseur, on utilise `xml_set_element_handler()` pour indiquer les fonctions qui doivent être déclenchées quand le parseur rencontre les balises ouvrante et fermante des éléments, soit ici `debutElement()` et `finElement()`. La fonction associée à une balise ouvrante doit accepter trois paramètres :

1. le parseur ;
2. le nom de l'élément rencontré ;
3. un tableau associatif contenant la liste des attributs XML contenus dans la balise ouvrante.

Quand on écrit cette fonction, on doit donc implanter l'action appropriée en tenant compte du nom de l'élément et des attributs. Dans notre cas, on affiche un message et on mémorise dans une variable globale le nom de l'élément rencontré.

```
function debutElement ($parser, $nom, $attrs)
{
  global $element_courant;
  $element_courant = $nom;

  echo "balise ouvrante de $nom\n";
}
```

L'inconvénient de cette fonction est que l'on ne peut pas étendre la liste des paramètres ou renvoyer une valeur. Le seul moyen de communiquer avec l'application est donc d'utiliser une variable globale (voir page 437) ce qui n'est pas très satisfaisant : nous verrons plus loin comment faire mieux avec la programmation objet.

La fonction déclenchée sur la balise fermante n'a pas de troisième argument (il n'y a pas d'attributs dans ces balises). Notre implantation se contente d'afficher un message rendant compte de l'événement rencontré.

```
function finElement ($parser, $nom)
{
  echo "balise fermante de $nom\n";
}
```

Le troisième type d'événement pris en compte dans ce module est la rencontre d'un nœud de texte. La fonction déclenchée est déclarée avec `xml_set_character_data_handler()`. La voici :

1. La liste complète des fonctions de cette API est donnée dans l'annexe C, page 500.

```
function donneesCaracteres ($parser, $chaine)
{
  global $tab_elements, $element_courant;

  if (trim($chaine) != "")
    $tab_elements[$element_courant] .= $chaine;
}
```

On reçoit en argument la chaîne de caractères constituant le contenu du nœud de texte. De tels nœuds sont très souvent constitués uniquement d'espaces ou de retours à la ligne, quand il servent uniquement à la mise en forme du document. On élimine ici ces espaces avec la fonction PHP `trim()` et on vérifie que la chaîne obtenue n'est pas vide. On sait alors qu'on a affaire au contenu d'un élément.

La fonction ne permet pas de savoir de quel élément il s'agit (rappelons que les éléments et le texte constituent des nœuds distincts dans l'arborescence d'un document XML, et sont donc traités séparément par le parseur). La seule solution est de s'appuyer sur une variable globale, `$element_courant` qui stocke le nom du dernier élément rencontré (voir la fonction `debutElement()`). On utilise ce nom pour mémoriser le contenu dans le tableau `$tab_elements`, lui aussi déclaré comme une variable globale.

Ce style de programmation, assez laborieux, est imposé par l'absence d'information sur le contexte quand une fonction est déclenchée. On ne sait pas à quelle profondeur on est dans l'arbre, quels sont les éléments rencontrés auparavant ou ceux qui qui vont être rencontrés après, etc. Cette limitation est le prix à payer pour l'efficacité du modèle d'analyse : le parseur se contente de parcourir le document une seule fois, détectant le marquage au fur et à mesure et déclenchant les fonctions appropriées.

Le code ci-dessous montre comment faire appel au module, en l'appliquant au document *Gladiator.xml* (voir page 318).

Exemple 8.13 *exemples/ExSAX.php* : *Exemple d'application du parseur*

```
<?php
  // Application des fonctions SAX
  require ("SAX.php");

  Header ("Content-type: text/plain");

  // Analyse du document
  $film = ParseFilm ("Gladiator.xml");

  // Affichage des données extraites
  while ( list ($nom, $val) = each ($film))
          echo "Nom : $nom Valeur : $val\n";
?>
```

On obtient alors le résultat suivant (noter que les noms d'éléments sont mis en majuscules par le parseur, option par défaut qui peut être modifiée) :

Exemple 8.14 *ResultatSAX.txt* : *Résultat de l'application du parseur*

```
balise ouvrante de FILM
balise ouvrante de TITRE
balise fermante de TITRE
balise ouvrante de ANNEE
balise fermante de ANNEE
balise ouvrante de CODEPAYS
balise fermante de CODEPAYS
balise ouvrante de GENRE
balise fermante de GENRE
balise ouvrante de REALISATEUR
balise fermante de REALISATEUR
balise fermante de FILM
Nom : TITRE Valeur : Gladiator
Nom : ANNEE Valeur : 2000
Nom : CODEPAYS Valeur : USA
Nom : GENRE Valeur : Drame
```

L'approche très simple employée ici se généralise difficilement à des documents XML plus complexes comprenant des imbrications d'éléments, comme par exemple un film avec son réalisateur et la liste de ses acteurs. Le fait de devoir mémoriser dans des variables globales le positionnement du parseur dans le document rend rapidement la programmation assez laborieuse.

Heureusement, une fonctionnalité très intéressante du parseur XML/PHP permet une intégration forte avec la programmation orientée-objet. Nous montrons dans ce qui suit comment utiliser cette fonctionnalité. C'est également l'occasion de revenir sur les notions de spécialisation et d'héritage de la programmation objet, présentées dans le chapitre 3.

8.3.3 Une classe de traitement de documents XML

La fonction `xml_set_object()` permet d'associer un parseur XML et un objet, avec deux effets liés :

- les « déclencheurs » seront pris parmi les méthodes de l'objet (ou, plus précisément, de la classe dont l'objet est une instance) ;
- ces déclencheurs ont accès aux variables d'état de l'objet, ce qui évite d'avoir recours aux variables globales.

On peut donc développer des classes, dédiées chacune à un type de document particulier (par exemple nos films), avec un style de programmation beaucoup plus élégant que celui employé précédemment. Chaque classe fournit, sous forme de méthodes, des fonctionnalités de création du parseur, de gestion des erreurs, de définition des déclencheurs, de lancement de l'analyse, etc. Afin d'essayer d'être –encore

une fois– le plus général possible, nous allons distinguer dans ces fonctionnalités celles, *génériques*, qui sont identiques pour tous les types de document de celles qui sont *spécifiques* à un type de document donné. Il serait dommage de dupliquer les premières autant de fois qu'il y a de types de documents, avec les inconvénients évidents en termes de maintenance et de modification qui en découlent. Il est donc souhaitable de recourir à la spécialisation objet qui permet de « factoriser » les fonctions génériques et de les rendre disponibles pour chaque classe traitant d'un type de document particulier.

Le principe consiste à créer une (super-)classe qui regroupe les fonctions génériques au traitement d'un document par SAX, et à créer, par spécialisation, une sous-classe de cette super-classe qui peut réutiliser ses fonctionnalités (héritage), les redéfinir (surcharge), et lui en ajouter d'autres (extension). Voici le code de la super-classe. Nous en décrivons plus bas les principales méthodes.

Exemple 8.15 *webscope/lib/SAX.php* : *Une classe générique de traitement d'un document XML*

```
<?php
/**
 * Classe générique d'appel aux fonctions SAX d'analyse de
 * documents XML.
 *  Cette classe doit être sous-typée pour chaque type de
 *  document
 *  Au niveau de la super-classe on se contente d'instancier le
 *  parseur,
 *  de gérer les messages d'erreur, et d'afficher au fur et à
 *  mesure
 *  le contenu du document analysé.
 */

class SAX
{
  private $parseur;
  protected $donnees; // Données caractères rencontrées en cours
      d'analyse
  protected $erreur_rencontree, $message; // Indicateur et
      message d'erreur

  // Constructeur de la classe
  function __construct ($codage)
  {
    // On instancie le parseur
    $this->parseur = xml_parser_create($codage);

    // On indique que les déclencheurs sont des méthodes de la
    // classe
    xml_set_object($this->parseur, &$this);

    // Tous les noms d'éléments et d'attributs seront traduits en
    // majuscules
```

```
    xml_parser_set_option($this->parseur, XML_OPTION_CASE_FOLDING,
        true);

    xml_set_element_handler ($this->parseur, "debutElement", "
        finElement");
    xml_set_character_data_handler ($this->parseur, "
        donneesCaracteres");

    $this->erreur_rencontree = 0;
  }

  // Méthode générique de traitement des débuts d'éléments
  protected function debutElement ($parseur, $nom, $attrs)
  {
    // On recherche si une méthode nommée "debut{NomElement}"
        existe
    if (method_exists ($this, "debut$nom")) {
      call_user_func (array($this, "debut$nom"), $parseur, $nom,
          $attrs);
    }
    else if (get_class($this) == "sax") {
      echo "Début d'élément : &lt;" . $nom . "&gt;\n";
    }

    // Effacement des données caractères
    $this->donnees = "";
  }

  // Méthode générique de traitement des fins d'élément
  protected function finElement ($parseur, $nom)
  {
    // On recherche si une méthode nommée "fin{NomElement}"
    // existe
    if (method_exists ($this, "fin$nom")) {
      call_user_func (array($this, "fin$nom"), $parseur, $nom);
    }
    else if (get_class($this) == "sax") {
      echo "Fin d'élément : &lt;/" . $nom . "&gt;\n";
    }
  }

  // Pour les données, on stocke simplement au fur et à mesure
  // dans la propriété $this->donnees

  protected function donneesCaracteres ($parseur, $chaine)
  {
    // On ne prend pas les chaînes vides ou ne contenant que des
    // blancs
    if (!empty($chaine))  $this->donnees .= $chaine;
  }
```

```
  // Méthode pour analyser le document
  public function parse($fichier)
  {
    // Ouverture du fichier
    if ( !($f = fopen($fichier, "r"))) {
      trigger_error ("Impossible d'ouvrir le fichier $fichier\n",
          E_USER_ERROR);
      return;
    }

    // Analyse du contenu du document
    while ($data = fread($f, 4096)) {
      if (!xml_parse($this->parseur, $data, feof($f))) {
        $this->erreur ("Erreur rencontrée ligne "
        . xml_error_string(xml_get_error_code($this->parseur))
        . "%de $fichier : "
        . xml_get_current_line_number($this->parseur));
        return;
      }
    }
    fclose ($f);
  }

  // Destructeur de la classe
  function __destruct()
  {
    xml_parser_free($this->parseur);
  }

  // Fonction retournant le message d'erreur
  public function messageErreur ($message)
  {
    return $this->message;
  }

  // Fonction indiquant si une erreur est survenue
  public function erreurRencontree () {
    return $this->erreur_rencontree;
  }

  // Fonction traitant une erreur
  function erreur ($message)
  {
    trigger_error ($message, E_USER_ERROR);
  }
}
?>
```

La classe comprend trois méthodes, `__construct()` (le constructeur), `parse()` et `__destruct()` (le destructeur), qui correspondent respectivement aux trois phases de l'analyse d'un document : création du parseur, ouverture du fichier et

analyse du document, enfin destruction du parseur. Ces trois méthodes sont valables quel que soit le document et seront donc reprises par héritage dans toutes les sous-classes. Outre la création du parseur, la principale caractéristique du constructeur est l'appel à la fonction `xml_set_object()` qui indique que le parseur est intégré à l'objet courant (`$this`) et que les déclencheurs sont des méthodes de cet objet.

La méthode `parse()` est implantée de manière un peu plus robuste que celle de notre premier exemple. En particulier :

1. le fichier est analysé par tranche de 4 096 octets pour éviter de charger en mémoire un document trop volumineux ; la fonction `xml_parse()` prend en troisième argument un indicateur de fin de fichier (ici retourné par la fonction `feof()`) qui indique qu'on a atteint la dernière tranche du document ;
2. une gestion d'erreurs est mise en place, en testant le code retour de `xml_parse()`, et en appelant le cas échéant des fonctions décrivant l'erreur : une description détaillée de toutes ces fonctions XML/PHP est donnée dans l'annexe C.

En plus de ces trois méthodes, la classe *SAX* propose une implantation « par défaut » des déclencheurs associés aux balises ouvrantes, fermantes et aux nœuds de texte (ou données caractères). Dans une implantation plus complète, la classe devrait également gérer les instructions de traitement, entités externes et autres catégories syntaxiques XML.

Que font ces méthodes ? Le principe est de regarder, pour un élément dont le nom *`NomElément`* est transmis en paramètre, si la classe comprend une méthode `debut`*`NomElément`*, une méthode `fin`*`NomElément`*, ou les deux. Si une telle méthode existe, elle est appelée. Sinon il y a deux possibilités :

1. soit l'objet `$this` est une instance de la super-classe *SAX*, et on affiche un message indiquant qu'une balise ouvrante ou fermante selon le cas a été rencontrée ;
2. soit l'objet est une instance d'une sous-classe, et dans ce cas l'absence d'une méthode pour l'élément indique que ce dernier doit être ignoré.

En d'autres termes, un objet de la classe *SAX* peut être utilisé comme notre premier module, page 335, pour afficher le marquage du document au fur et à mesure qu'il est rencontré. Un objet d'une sous-classe de *SAX* en revanche doit explicitement fournir des méthodes de traitement des éléments, sinon il ne se passe rien.

Les méthodes `debutElement()` et `finElement()` s'appuient sur les fonctions PHP suivantes :

- `method_exists()` qui teste si une méthode est définie pour un objet,
- `call_user_func()` qui permet de déclencher l'exécution d'une méthode pour un objet [2],

2. En PHP, on désigne une fonction par son nom, et une méthode par un tableau contenant une paire (objet, nom de la méthode).

- enfin `get_class()` qui renvoie, en minuscules, le nom de la classe dont l'objet courant est une instance.

Voici la mise en œuvre de ces fonctions dans la méthode `debutElement()` :

```
protected function debutElement ($parseur, $nom, $attrs)
{
  // On recherche si une méthode nommée "debut{NomElement}"
      existe
  if (method_exists ($this, "debut$nom")) {
    call_user_func (array($this, "debut$nom"), $parseur, $nom,
        $attrs);
  }
  else if (get_class($this) == "sax") {
    echo "Début d'élément : &lt;" . $nom . "&gt;\n";
  }

  // Effacement des données caractères
  $this->donnees = "";
}
```

Pour bien comprendre l'intérêt de cette conception qui peut sembler compliquée au premier abord, il suffit de constater que la création d'un analyseur pour un type donné de document se résume maintenant à créer une sous-classe de *SAX* et à placer dans cette sous-classe des méthodes `debut`*`Nom`* et `fin`*`Nom`* spécifiques aux noms d'éléments du type de document. Voici la classe *SAXFilm*, sous-classe de *SAX* (noter le mot-clé `extends`), qui permet d'analyser nos documents contenant des films (voir l'exemple du document *KillBill.xml*, page 331).

Exemple 8.16 *webscope/application/classes/SAXFilm.php* : *Spécialisation de la classe pour traiter des films*

```
<?php

/**
 *  Classe d'analyse avec le parseur SAX d'un document XML
 *  représentant un film avec ses acteurs.
 */

require_once ("SAX.php");

class SAXFilm extends SAX
{
  // Déclaration des variables stockant le résultat de l'analyse
  // private $films, $nb_films, $i_acteur;

  // Constructeur
  function SAXFilm ($codage)
  {
    // On appelle le constructeur de la super-classe
    parent::__construct($codage);
```

```
    // On initialise les variables de la sous-classe
    $this->films = array();
    $this->nb_films = 0;
    $this->i_acteur = 0;
}

// On définit les déclencheurs pour les différents éléments de
    FILM

protected function finTITRE ($parseur, $nom)
{ $this->films[$this->nb_films]['titre'] = $this->donnees; }
protected function finANNEE ($parseur, $nom)
{ $this->films[$this->nb_films]['annee'] = $this->donnees; }
protected function finCODE_PAYS ($parseur, $nom)
{ $this->films[$this->nb_films]['code_pays'] = $this->donnees; }
protected function finGENRE ($parseur, $nom)
{ $this->films[$this->nb_films]['genre'] = $this->donnees; }
protected function finRESUME ($parseur, $nom)
{ $this->films[$this->nb_films]['resume'] = $this->donnees; }

// Pour le metteur en scène
protected function debutREALISATEUR ($parseur, $nom, $attr)
{
    $this->films[$this->nb_films]['nom_realisateur']
      = $attr['NOM'];
    $this->films[$this->nb_films]['prenom_realisateur']
      = $attr['PRENOM'];
    $this->films[$this->nb_films]['annee_realisateur']
      = $attr['ANNEE_NAISSANCE'];
}

// Pour un acteur
protected function debutACTEUR ($parseur, $nom, $attr)
{
    $this->films[$this->nb_films]['nom'][$this->i_acteur]
          = $attr['NOM'];
    $this->films[$this->nb_films]['prenom'][$this->i_acteur]
          = $attr['PRENOM'];
    $this->films[$this->nb_films]['annee_naissance'][$this->
        i_acteur]
          = $attr['ANNEE_NAISSANCE'];
    $this->films[$this->nb_films]['nom_role'][$this->i_acteur]
          = $attr['NOM_ROLE'];
    $this->i_acteur++;
}

// Au début d'un élément FILM: on incrémente le compteur de
// films,
// on initialise le compteur des acteurs et les tableaux
protected function debutFILM ($parseur, $nom)
{
  $this->nb_films++;
```

```
    $this->films[$this->nb_films] = array();
    $this->films[$this->nb_films]['nom'] = array();
    $this->films[$this->nb_films]['prenom'] = array();
    $this->films[$this->nb_films]['annee_naissance'] = array();
    $this->films[$this->nb_films]['nom_role'] = array();
    $this->i_acteur = 0;
  }

  /********** Partie publique  ********************/

  // Récupération d'un film
  public function getNbFilm ()
  {    return $this->nb_films; }

  public function getFilm ($id_film)
  {
    if ($id_film > 0 and $id_film <= $this->nb_films)
      return $this->films[$id_film];
    else
      {
        // Mais non, cet id n'existe pas!!!
        trigger_error("Index incorrect pour accéder à un film:
            $id_film");
        return array();
      }
  }
}
```

Cette sous-classe comprend essentiellement des méthodes pour traiter les éléments spécifiques d'un document <Film> : tout ce qui relève de la création et de l'utilisation du parseur est hérité de la classe *SAX*, et donc automatiquement disponible pour les objets de la classe *SAXFilm*.

Dans le cas des éléments `titre`, `annee`, `code_pays`, etc., la chaîne à extraire est le contenu de l'élément, autrement dit le nœud de type texte situé sous l'élément dans l'arborescence. On récupère cette chaîne en déclenchant une fonction quand on rencontre la balise fermante. On sait alors que la chaîne est mémorisée dans l'attribut `$this->donnees`, et on la stocke dans le tableau des films.

Dans le cas des éléments `Realisateur` ou `Acteur`, les chaînes de caractères à extraire sont les attributs de l'élément. On déclenche cette fois une fonction sur la balise ouvrante qui se charge de recopier le tableau des attributs dans `$tab_film`.

Au moment de l'analyse du document, l'existence de ces méthodes sera détectée par `debutElement()` ou `finElement()`, héritées de la classe *SAX* ; elles seront donc automatiquement invoquées au moment approprié.

Une dernière remarque : le constructeur de la classe *SAXFilm* initialise les données qui sont propres aux films. Il faut également appeler le constructeur de la super-classe *SAX* puisque ce dernier a en charge la création du parseur. Cet

appel du constructeur des parents dans la hiérarchie de spécialisation n'est pas automatique en PHP.

Voici pour finir le script associé au formulaire qui permet de transmettre un document XML pour l'insérer dans la base MySQL. La création d'un objet `$filmXML`, instance de la classe *SAXFilm*, permet d'analyser de manière complètement transparente le contenu du document et de récupérer les valeurs décrivant un film[3]. Pour le contrôle des données et l'insertion dans la base, on fait appel aux fonctions du modèle implantées dans *Film.php* par extension de la classe `TableBD`.

```
function import ()
{
  // Définition du titre
  $this->vue->titre_page = "Import de données XML";

  // Contrôle de la session
  $this->controleAcces();

  // Création du parseur pour les films
  $film_XML = new SAXFilm("ISO-8859-1");

  // Analyse du document
  if (!isSet($_FILES['fichierXML'])) {
    $this->vue->contenu = "Il faut indiquer un fichier XML <br
        />";
    echo $this->vue->render("page");
    return;
  }
  else {
    $film_XML->parse($_FILES['fichierXML']['tmp_name']);
  }

  // On insère, si aucune erreur n'est survenue
  if (!$film_XML->erreurRencontree()) {
    $film = $film_XML->getFilm(1);

    // On instancie le modèle "Film" (voir le contrôleur Saisie
        )
    $tbl_film = new Film ($this->bd);

    // On crée un objet à partir des données du tableau HTTP
    $tbl_film->nouveau($film);

    // Contrôle des valeurs reçues
    $message = $tbl_film->controle() ;

    // Vérification des valeurs récupérées dans le document
    if ($message) {
```

3. Ce code fonctionne également pour un document contenant *plusieurs* films, mais pour simplifier nous ne traitons que le premier film rencontré.

```
            // Il y a un problème. Affichage du formulaire de saisie
            // avec les valeurs du tableau
            $this->vue->contenu = "Problème: $message<br/>"
            . $tbl_film->formulaire (TableBD::INS_BD,
                  "saisie", "inserer");
        }
        else {
            // Insertion, avec la fonction du modèle
            $tbl_film->insertion();
            // On affiche ce qu'on vient d'insérer
            $this->vue->id_film = $tbl_film->id;
            $this->vue->titre = $tbl_film->titre;
            $this->vue->annee = $tbl_film->annee;
            $this->vue->realisateur = $tbl_film->nomRealisateur();
            $this->vue->fichier_xml =  $_FILES['fichierXML']['name'];
            $this->vue->setFile("contenu", "xml_import.tpl");
        }
    }
    echo $this->vue->render("page");
}
```

En résumé, nous avons vu dans cette section comment écrire un module d'analyse de documents XML simple mais limité, puis comment utiliser des aspects avancés de la programmation orientée-objet pour obtenir des outils beaucoup plus puissants, comprenant une bonne partie de code réutilisable en toutes circonstances. On peut noter que cette programmation permet notamment d'éviter le recours aux variables globales, qui mène rapidement à des scripts difficilement compréhensibles.

8.4 MISE EN FORME DE DOCUMENTS AVEC XSLT

Pour conclure cette présentation des possibilités d'utilisation de documents XML en association avec MySQL/PHP, nous montrons comment transformer un document obtenu par export d'une base MySQL en un format quelconque avec le langage de transformation XSLT. La motivation est la même que celle déjà explorée avec les *templates* : on voudrait pouvoir séparer complètement le traitement du *contenu*, ici obtenu par de la programmation PHP/MySQL, et la *présentation* de ce contenu. Dans le cas des *templates* on utilisait des « modèles » HTML comprenant des « entités » et on remplaçait ensuite, par programmation, ces entités par des chaînes de caractères PHP construites dynamiquement.

La technique XML/XSLT est comparable, avec deux différences notables :

- les modèles de *templates* sont remplacés par des *règles* et autres instructions XSLT ;
- au lieu de placer le contenu dans une chaîne de caractères, on le représente de manière structurée dans un document XML.

Contrairement aux systèmes de *templates*, des solutions *ad hoc* dépendantes de la bonne volonté d'un petit groupe de programmeurs, XSLT est un langage officiellement promu et défini par le *World Wide Web Consortium*, conçu pour offrir toute la puissance nécessaire à la publication de contenus complexes, et offrant incontestablement de solides garanties de pérennité. De plus, pour un même type de document XML, il est possible de définir plusieurs programmes XSLT, chacun correspondant à un format de sortie différent. XSLT est une solution à considérer si on construit un site publiant un même contenu à destination de plusieurs médias.

En contrepartie, XSLT est un langage complexe à apprendre et à maîtriser, et qu'avant de s'engager dans une solution technique comprenant des langages aussi différents que HTML, PHP, SQL, XML, XSLT, etc., il faut réfléchir sérieusement à ses avantages et inconvénients.

La présentation qui suit a essentiellement pour objectif de vous permettre d'apprécier les caractéristiques d'une telle solution, à charge pour vous ensuite de faire vos choix en connaissance de cause. Il exclu de se lancer ici dans une présentation du langage XSLT. Les quelques exemples présentés se veulent suffisamment intuitifs pour saisir les principes de base.

8.4.1 Quelques mots sur XSLT

Un programme XSLT est un document XML, constitué d'un élément racine `<xsl:stylesheet>`, comprenant lui-même d'autres éléments parmi lesquels ceux qui correspondent à des instructions XSLT sont tous préfixés par « `xsl:` ». Les principaux éléments XSLT sont les *règles*, de nom `<xsl:template>`.

Un *programme XSLT* s'applique à un *document source* XML en entrée et produit un *document résultat* (qui est lui aussi du XML). La production du document résultat s'appuie sur les règles du programme. Intuitivement, le processeur XSLT commence par parcourir le document source, lequel est considéré comme une arborescence de nœuds. Pour chaque nœud, le processeur regarde si une des règles du programme XSLT s'applique au nœud. Si c'est le cas, un fragment du document résultat décrit dans la règle va être produit. L'assemblage de tous les fragments produits au cours de l'exécution d'un programme constitue le document résultat complet.

Nous allons présenter un programme XSLT qui s'applique aux documents XML exportés de la base *Films* (voir par exemple *KillBill.xml*, page 331) et les transforme en un document HTML. Voici le programme XSLT « principal », avec une seule règle.

Exemple 8.17 *La règle XSLT pour la production d'une page HTML*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:include href="Film.xsl"/>
  <xsl:output method="html"
              encoding="ISO-8859-1"/>
```

```
  <xsl:template match="/">

   <!-- Entête de la page                                    -->
   <html>
    <head>
      <title>Page produite avec XSLT</title>
      <link rel="stylesheet" href="films.css" type="text/css"/>
    </head>
    <body>

    <center><h1>Résultat de la mise en forme XSLT</h1></center>

    Voici la liste des films, mis au format HTML.

    <!-- Mise en forme du document XML -->
    <xsl:apply-templates/>

   </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

Ce programme est bien un document XML, avec un élément racine `<xsl:stylesheet>`, contenant deux instructions `<xsl:include>` et `<xsl:output>` sur lesquelles nous reviendrons plus tard, et une règle `<xsl:template>`. Le principe d'un tel programme est d'exécuter les instructions XSLT et de les remplacer par le résultat de cette exécution qui doit être un fragment XML bien formé. Quand toutes les instructions ont été traitées, il reste un document XML résultat sans traces d'instructions XSLT. Ici le « dialecte » choisi pour la sortie est du (X)HTML.

L'attribut `match` de la règle indique les nœuds du document source pour lesquels elle va être déclenchée. La valeur de cet attribut doit être une expression d'un autre langage, XPath, qui permet d'exprimer des chemins dans une arborescence XML. L'expression « / » indique en l'occurrence que la règle s'applique à la *racine du document* XML. Quand le processeur va rencontrer ce nœud (il s'agit du premier rencontré), la règle est déclenchée et produit la version intermédiaire suivante du document résultat.

Exemple 8.18 *Regle1.xml* : *Résultat intermédiaire après application de la première règle*

```
<html>
 <head>
   <title>Page produite avec XSLT</title>
   <link rel="stylesheet" href="films.css" type="text/css"/>
 </head>
 <body>

 <center><h1>Mise en forme XSLT</h1></center>
```

```
    Voici la liste des films, mis au format HTML.

    <!-- Mise en forme du document XML -->
    <xsl:apply-templates/>

  </body>
</html>
```

Ce document est *déjà* un document XML bien formé, mais pas encore un document HTML. Il contient encore une instruction XSLT, <xsl:apply-templates>, ce qui indique que le traitement n'est pas fini. Le processeur XSLT va donc continuer à parcourir le document source, en cherchant les règles qui s'appliquent à ses nœuds et en déclenchant les règles appropriées.

Dans notre premier programme nous avions une instruction <xsl:include>. Comme son nom l'indique, elle inclut un autre programme XSLT, *Film.xsl*, qui contient une unique autre règle.

Exemple 8.19 *La règle XSLT pour la présentation d'un film*

```
<?xml version="1.0" encoding="iso-8859-1"?>

<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html" indent="yes"/>

  <xsl:template match="Film">
          <h1><i><xsl:value-of select="titre"/></i></h1>

          <!-- Genre, pays, et année du film      -->
          <xsl:value-of select="genre"/>,
          <i> <xsl:value-of select="code_pays"/></i>,
          <xsl:value-of select="annee"/>.

          <!-- Auteur du film  -->
          Mis en scène par
            <b><xsl:value-of
               select="concat(realisateur/prenom, ' ', realisateur
                   /nom)"/>
            </b>

        <h3>Acteurs</h3>
        <xsl:for-each select="Acteur">
           <b><xsl:value-of select="concat(prenom,' ',nom)"/></b>
                :
                 <xsl:value-of select="nom_role"/><br/>
        </xsl:for-each>

        <!-- Résumé du film  -->
        <h3>Résumé</h3>
```

```
            <xsl:value-of select="resume"/>

  </xsl:template>
</xsl:stylesheet>
```

L'attribut `match` indique que cette seconde règle se déclenche sur un nœud <`Film`>. Ce déclenchement produit un fragment qui constitue une représentation HTML des différentes données constituant le film rencontré. La page HTML résultat contiendra, s'il y a trois films, les trois fragments successivement produits par exécution de cette règle. Les données sont extraites du document source avec l'instruction <`xsl:value-of`>. Par exemple :

- <`xsl:value-of select="titre"`> insère dans le résultat le contenu de l'élément <`titre`>, fils de l'élément <`Film`> ;
- <`xsl:value-of select="Realisateur/@nom"`> insère dans le résultat la valeur de l'attribut `nom` de l'élément <`Realisateur`>, fils de l'élément <`Film`> ;

On peut noter que XSLT permet également d'effectuer des boucles sur un ensemble d'éléments, ici les acteurs, avec l'instruction <`for-each`>. À chaque fois que le processeur évalue une de ces instructions, elle est remplacée par le fragment XML produit. Quand toutes les instructions ont été traitées, on obtient un document HTML.

Voici par exemple ce que donne l'exécution de cette seconde règle, à partir du document intermédiaire *Regle1.xml*, page 350, appliqué au document source *KillBill.xml* page 331. On constate que le `xsl:apply-templates` est remplacé par le fragment XHTML produit :

Exemple 8.20 *Regle2.xml : Après application de la deuxième règle*

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
<title>Page produite avec XSLT</title>
<link rel="stylesheet" href="films.css" type="text/css"/>
</head>
<body>
<center><h1>Résultat de la mise en forme XSLT</h1></center>

    Voici la liste des films, mis au format HTML.

  <h1><i>Kill Bill</i></h1>Drame, <i>USA</i>, 2003.

  Mis en scène par <b>Quentin Tarantino</b>.
  <h3>Acteurs</h3>

   <b>Uma Thurman</b>: La mariée, alias "Black Mamba"<br/>
   <b>Lucy Liu</b>: O-Ren Ishii<br/>
   <b>David Carradine</b>:  Bill<br/>
```

```
   <b>Michael Madsen</b>: Budd / Sidewinder<br/>
   <b>Daryl Hannah</b>: Elle Driver<br/>

  <h3>Résumé</h3>
   Au cours d'une cérémonie de mariage en plein désert, ...
</body>
</html>
```

La transformation est terminée, et ce document HTML est prêt à être affiché dans n'importe quel navigateur. En résumé, XSLT est un langage qui permet de produire un document XML par assemblage de fragments contenus dans des règles, et en incluant dans ces fragments des parties extraites d'un document source. Voyons maintenant comment appliquer une telle transformation avec PHP.

8.4.2 Application d'un programme XSLT avec PHP

On peut envisager deux possibilités pour effectuer la transformation XSLT : côté serveur ou côté client. Pour le côté serveur, PHP fournit une interface fonctionnelle avec le processeur XSLT Comme pour SAX, cette interface permet de créer un processeur, et d'appliquer un programme XSLT à un document source. Le programme, le document source et le document résultat peuvent être soit des chaînes de caractères, soit des fichiers.

Ce processeur n'est pas toujours installé, et la transformation côté client est plus facile à metre en œuvre. Elle consiste à transmettre le document XML et le programme XSLT au navigateur et à laisser ce dernier effectuer la transformation. Ce n'est possible qu'avec un navigateur doté d'un processeur XSLT, comme les versions raisonnablement récentes de Firefox, Safari ou Internet Explorer. Il suffit alors d'ajouter une instruction de traitement

```
<?xml-stylesheet href='programme' type='text/xsl'?>
```

dans le prologue du document XML pour indiquer au processeur XSLT le programme à appliquer.

```
// On envoie l'en-tête HTTP, et le prologue du document XML
 Header ("Content-type: text/xml");
 echo "<?xml version=\"1.0\" encoding=\"iso-8859-1\"?>\n\n";

 // Mise en forme selon le choix de l'utilisateur
 if ($_POST['format'] == "XML") {
   // On sort le XML brut
   echo "<Films>\n$document</Films>\n";
 }
 else {
   // On applique une transformation XSLT. Il suffit d'ajouter
   // une instruction pour que le navigateur en tienne compte
   // et applique la transformation Film.xsl
```

```
        echo "<?xml-stylesheet href='./xsl/Film.xsl' type='text/xsl
            '?>\n"
                    . "<Films>\n$document</Films>\n";;
    }
```

On voit que l'instruction de traitement est placée entre le prologue et le corps du document quand on a demandé un export en HTML. Dans ce cas, le document XML *et* le programme XSLT sont transmis au navigateur qui effectue la transformation et affiche directement le résultat comme le montre la figure 8.2. On utilise PHP uniquement pour la production du document XML, et la mise en forme (ici HTML) est obtenue avec XSLT. Il serait très facile de créer de nouveaux programmes XSLT applicables au même document pour produire, par exemple une version pour des téléphones mobiles, une version VoiceXML (sortie vocale), une version RSS, etc. Notez que les personnes qui réalisent ces programmes XSLT n'ont aucun besoin de connaître ni la structure de la base (ou même son existence), ni PHP, ni l'architecture du site.

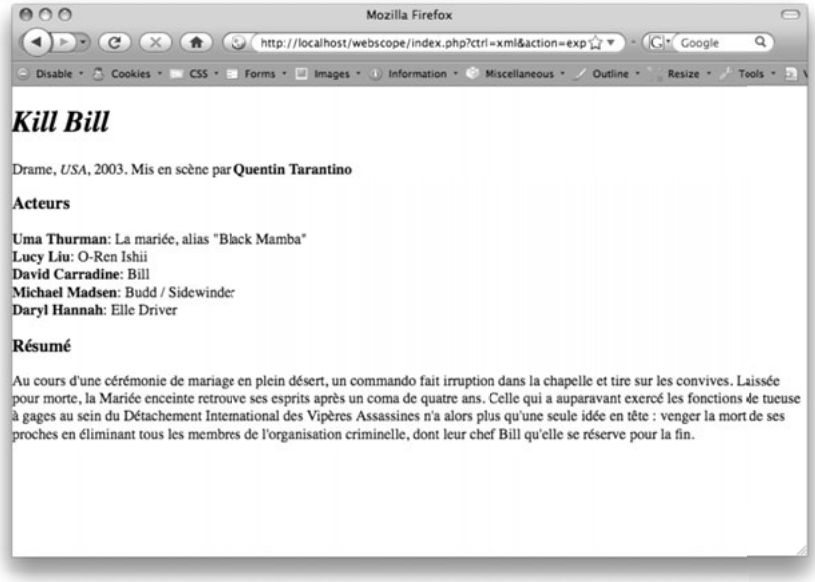

Mozilla Firefox
http://localhost/webscope/index.php?ctrl=xml&action=exp
Google
Disable
Cookies
CSS
Forms
Images
Information
Miscellaneous
Outline
Resize
Tools
Kill Bill
Drame, USA, 2003. Mis en scène par Quentin Tarantino
Acteurs
Uma Thurman: La mariée, alias "Black Mamba"
Lucy Liu: O-Ren Ishii
David Carradine: Bill
Michael Madsen: Budd / Sidewinder
Daryl Hannah: Elle Driver
Résumé
Au cours d'une cérémonie de mariage en plein désert, un commando fait irruption dans la chapelle et tire sur les convives. Laissée pour morte, la Mariée enceinte retrouve ses esprits après un coma de quatre ans. Celle qui a auparavant exercé les fonctions de tueuse à gages au sein du Détachement International des Vipères Assassines n'a alors plus qu'une seule idée en tête : venger la mort de ses proches en éliminant tous les membres de l'organisation criminelle, dont leur chef Bill qu'elle se réserve pour la fin.

Figure 8.2 — Résultat de la transformation XSLT

L'intérêt de recourir à cette solution est essentiellement de pouvoir choisir dynamiquement entre plusieurs programmes XSLT au moment de la publication des données. Cela peut permettre de personnaliser la présentation en fonction du navigateur, du média (ordinateur, téléphone, PDA, ...), ou des souhaits d'un utilisateur particulier. Imaginons par exemple un site qui gère un catalogue de produits (disons, des livres), et plusieurs fournisseurs, disposant chacun de leur propre site web, et souhaitant y publier avec leurs propres normes graphiques une partie de ce catalogue. La programmation MySQL/PHP permet facilement d'extraire les données de la base, au format XML, et il reste à créer autant de programmes XSLT qu'il y a de présentations possibles.

TROISIÈME PARTIE

Compléments

9

Introduction au *Zend Framework*

Ce chapitre est une introduction au *Zend Framework* (abrégé en ZF), un ensemble de composants logiciels développé en *Open Source* (sous licence *Free BSD*) à l'initiative de la société Zend qui distribue l'interpréteur PHP et commercialise de nombreux outils pour la réalisation d'applications web professionnelles.

Le ZF est un ensemble extrêmement riche de classes orientées-objet dont l'ambition est de fournir un support au développement d'applications PHP complexes, dans tous les domaines. Nous allons nous intéresser essentiellement ici à la réalisation d'applications basées sur MySQL, en revisitant notamment le *pattern* MVC décrit dans le chapitre 6, tel qu'il est implanté dans le ZF.

Le ZF est un projet relativement récent (2005) qui se développe à grande échelle. Comme tous les outils de ce type (par exemple le *framework* STRUTS pour Java), sa prise en main peut s'avérer délicate car on est submergé de concepts qui peuvent paraître barbares au néophyte. Si vous avez bien assimilé le MVC « léger » présenté précédemment et les principes de la programmation objet, l'introduction qui suit doit vous éviter la phase la plus pénible de l'apprentissage. L'effort en vaut la peine car, pour des projets importants, l'utilisation d'un framework facilite bien les choses. Le choix de présenter le ZF ne doit d'ailleurs pas s'interpréter comme un jugement de valeur. Il existe d'autres frameworks très recommandables (pour n'en citer qu'un, *Symphony* semble très apprécié), mais vous devez à l'issue de la lecture être capable de vous débrouiller pour explorer d'autres pistes.

> **REMARQUE –** J'ai pris pour point de départ de cette introduction quelques documents trouvés sur le Web. Je remercie les auteurs, parmi lesquels Julien Pauli, qui a déposé quelques précieux tutoriaux sur le site *Developpez.com*.

Le chapitre s'appuie sur la présentation d'une application, appelée ZSCOPE, qui utilise ponctuellement la base MySQL sur les films, développée dans les chapitres

précédents, et propose quelques fonctionnalités implantées avec le ZF. Vous pouvez en récupérer le code sur le site du livre.

9.1 MISE EN ROUTE

La première chose à faire est d'installer le ZF et notre application. Une fois que cela fonctionne, lisez la fin de cette section qui présente l'organisation du code et les conventions d'écriture du ZF. Ce n'est pas évident, surtout au début, et il vaut mieux comprendre d'emblée comment cela fonctionne.

9.1.1 Installation d'une application ZF

La tâche la plus simple est d'installer le Zend Framework. Allez sur le site *http://framework.zend.com/* et récupérez l'archive contenant l'ensemble des composants. La version courante au moment où ces lignes sont écrites est la 1.6, et nous pouvons nous contenter de la version minimale, sans les utilitaires JavaScript *Dojo*.

Décompressez l'archive. On obtient un répertoire *ZendFramework-1.6* contenant un sous-répertoire *library*. Ce sous-répertoire contient lui-même un répertoire *Zend* qui est la racine de l'ensemble des classes du framework. Copiez *ZendFramework-1.6* sur votre disque, à un endroit accessible au serveur web. Vous pouvez aussi le renommer. Dans notre cas, il s'agit de *ZF*, placé dans */usr/local/share*.

> **REMARQUE –** Le ZF est très flexible et tout ou presque tout (noms de répertoires, organisation des répertoires, etc.) est paramétrable. Nous allons être assez directif pour éviter de nous embrouiller avec une longue liste d'options. Une fois que vous aurez compris les principes, vous pourrez vous lancer dans les variantes si vous le souhaitez.

Maintenant, récupérez l'archive du ZSCOPE, sur notre site. Décompressez-le et placez le répertoire racine *zscope* dans *htdocs*. Une autre possibilité est de récupérer le code sur le site CVS de *http://webscope.cvs.sourceforge.net* pour pouvoir faire des modifications et améliorer le ZSCOPE. Un défi possible, si plusieurs lecteurs sont intéressés, est de refondre le WEBSCOPE développé avec notre MVC personnel, en une nouvelle version entièrement basée sur le *Zend Framework*. Si vous êtes tentés, allez voir sur le site de SourceForge l'état des choses au moment où vous lisez ce livre.

Après cette installation initiale, il n'est pas encore possible d'accéder à ZSCOPE avec l'URL *http://localhost/zscope*, car le ZF s'appuie sur un principe de redirection et de réécriture des requêtes HTPP dont il faut au préalable s'assurer le bon fonctionnement.

9.1.2 Redirection des requêtes avec le ZF

L'application ZSCOPE contient deux sous-répertoires :

1. un répertoire *application*, avec tout le code d'une application MVC, à savoir les contrôleurs, les modèles et les vues ; voir plus loin pour des détails ;
2. un répertoire *www* destiné à contenir la partie *publique* de l'application, autrement dit tous les fichiers qui peuvent être directement référencés par une URL, et seulement ceux-là.

L'idée est que tous les fichiers qui peuvent être directement envoyés à un navigateur (images, PDF, CSS, Javascript, etc.) sont dans *www*. Le code de l'application elle-même n'est pas dans *www* mais dans *application*, afin d'interdire qu'on puisse accéder avec un navigateur à ce code, pour limiter les risques de fuite.

Le seul fichier PHP qui se trouve dans *www* est *index.php*. C'est lui qui charge les parties de l'application nécessaires à la satisfaction d'une requête HTTP, et *toutes* ces requêtes lui sont adressées sans exception. Voilà pour les principes. Leur mise en œuvre suppose un peu de configuration.

Configuration d'un hôte virtuel

À la base, le site est accessible à l'URL *http://localhost/zscope/www* (au besoin remplacez *localhost* par le nom de votre machine). On peut s'en contenter sur un site de test, mais rien n'empêche de tenter d'accéder à *http://localhost/zscope/application*, ce que l'on veut éviter.

Le serveur Apache permet la définition d'*hôtes virtuels* qui correspondent à un site particulier, sur une machine particulière. On peut avoir plusieurs hôtes virtuels pour un même serveur web, ce qui mène à engendrer (virtuellement) plusieurs espaces de noms correspondant à la même adresse IP.

Pour définir un hôte virtuel, éditez le fichier *httpd.conf* et ajoutez le bloc d'instructions suivant :

```
NameVirtualHost *:80

<VirtualHost *:80>
ServerName zscope.local
DocumentRoot /Applications/MAMP/htdocs/zscope/www
</VirtualHost>
```

Il faut de plus associer le nom `zscope.local` à l'IP 127.0.0.1 de la machine locale (si vous travaillez sur la machine locale). Pour cela on ajoute la ligne suivante dans */etc/hosts* (sur Linux ou Mac OS) ou `c:/windows/system32/drivers/etc/hosts` (Windows). Maintenant, redémarrez Apache, et vous devriez pouvoir accéder au site ZSCOPE à l'URL *http://zscope.local*. Toute l'arboresscence du site externe à *www*, et en particulier *application*, est devenue invisible et inaccessible.

Tout rediriger vers index.php

La seconde partie de la configuration consiste à effectuer une réécriture des requêtes HTTP pour les rediriger systématiquement vers le fichier *index.php*. On utilise pour cela le module *mod_rewrite* d'Apache, et des instructions placées dans un fichier *.htaccess*.

Vérifiez que le module *mod_rewrite* est bien actif pour votre serveur Apache en cherchant la ligne suivante dans *httpd.conf*, sinon ajoutez-la.

```
LoadModule rewrite_module modules/mod_rewrite.so
```

Ensuite, vous pouvez vous appuyer sur le fichier *.htaccess* présent dans *www*. Voici son contenu :

```
# Réécriture des requêtes
RewriteEngine On
RewriteCond %{REQUEST_URI} !\.(js|ico|gif|jpg|png|css)$
RewriteRule .* index.php
```

Un fichier *.htaccess* contient des instructions à destination du serveur Apache, propres aux fichiers contenus dans le répertoire courant. Ici, on demande à ce que toutes les URL autres que celles référençant des fichiers à transmettre directement au navigateur (images, Javascript, CSS, PDF, etc.) soient redirigées vers *index.php*. Essayez par exemple d'accéder à l'adresse *http://zscope.local/essai.php*. Si votre configuration fonctionne, le serveur vous redirigera vers *index.php*.

Cela supprime en grande partie les messages `404 Not Found` renvoyés par le serveur quand une URL ne correspond pas à une ressource existante. Ce message ne peut plus apparaître que pour les URL non redirigées, comme par exemple un fichier image qui n'existe pas.

Si tout va bien, vous devriez cette fois accéder à la page d'accueil et voir l'affichage de la figure 9.1. Comme vous le voyez, il s'agit du WEBSCOPE, dans un état embryonnaire, puisqu'aucune fonction n'est réalisée. Les contrôleurs et actions existants sont de simples illustrations des composants du ZF.

> **REMARQUE –** Si l'application ZSCOPE n'est pas à la racine de votre serveur web (par exemple si vous y accédez avec *http://localhost/zscope*), vous devez définir le paramètre `base_url` dans le fichier *application/config.ini* (sur le même exemple, le paramètre doit être `zscope`). Sinon, les images et feuilles de style CSS ne seront pas trouvées.

9.1.3 Organisation et conventions

Maintenant, jetons un coup d'œil à l'ensemble de l'organisation du site, résumée dans la figure 9.2. Elle suit les règles par défaut du Zend Framework. Bien que cela puisse sembler inutilement compliqué de prime abord, il faut se souvenir qu'on cherche à gérer des applications larges et complexes. Le découpage très prononcé en une hiérarchie de répertoires assez profonde a l'avantage de faciliter la localisation des différentes parties d'une application.

ZScope – Pratique de MySQL et PHP
http://zscope.local/
Le Webscope
Pratique de MySQL et PHP | Philippe Rigaux
Page d'accueil
Ce site correspond à une introduction au Zend Framework!
Ce site ne propose pas une application réelle, mais juste quelques exemples de programmation avec le Zend Framework, et en particulier des aspects MVC.
Reportez-vous au livre pour obtenir des explications sur la réalisation des différentes fonctionnalités et l'architecture générale du site. Chaque contrôleur correspond à une section du site:
1. Le contrôleur index montre simplement le présent texte.
2. Le contrôleur mvc développe quelques aspects du MVC Zend.
3. Le contrôleur model introduit les modèles Zend.
4. Le contrôleur view est consacré à la Zend View
5. Enfin le contrôleur bd décrit l'interface avec MySQL.
Il reste le contrôleur inscription qui vous invite à réalise le WebScope avec Zend. A vous de jouer ...
Menu
Accueil
Le MVC Zend
La vue Zend
Le modèle Zend
Bases de données
A vous de jouer...

Figure 9.1 — Page d'accueil du ZSCOPE

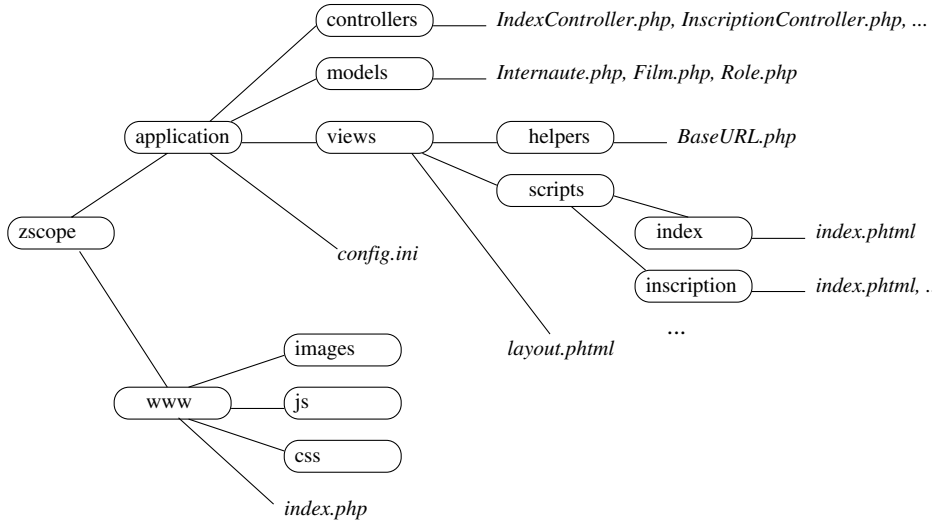


Figure 9.2 — Organisation (minimale) du code pour une application Zend

Vous pouvez déjà remarquer qu'il s'agit d'une extension de la structuration adoptée pour notre MVC simplifié utilisé pour la réalisation du WEBSCOPE[1]. Une

1. Il serait plus juste d'admettre que notre MVC est une simplification radicale du ZF.

différence est le répertoire *www* qui ne contient que les fichiers qui peuvent être référencés par une URL dans un navigateur. Tout le reste se trouve dans *application*. Comme dans notre MVC, ce dernier contient trois sous-répertoires correspondant respectivement aux contrôleurs, aux modèles et aux vues. Il contient également un fichier de configuration, *config.ini*.

Les fichiers des contrôleurs Zend sont nommés *Nom*`Controller`, où *Nom* représente le nom du contrôleur, commençant par une majuscule. Ici, nous avons les contrôleurs `index`, `inscription`, etc., correspondant aux fichiers *IndexController.php*, *InscriptionController.php*, etc. Comme dans notre MVC, un contrôleur est une classe constituée d'*actions*, méthodes dont le nom se termine par `Action`. Nous y revenons page 373.

Les modèles sont des classes PHP. Les classes représentant des données persistantes sont construites selon une *mapping* objet-relationnel proposé par le ZF qui permet de naviguer dans la base sans effectuer de requêtes SQL. Voir page 379.

Enfin, le répertoire *views* contient les vues. Sa structuration est un peu plus compliquée. Les fragments de pages HTML sont dans le sous-répertoire *scripts*, et on trouve encore un sous-répertoire pour chaque contrôleur. Les fichiers de vues ont tendance à proliférer, d'où une structuration en répertoires. Les fichiers ont pour extension *.phtml* car il s'agit d'un mélange de PHP et de HTML. Notez également dans *scripts* la présence du fichier *layout.phtml*, qui contient la mise en forme graphique du site.

Dans ce qui suit, référez-vous au schéma de la figure 9.2 pour retrouver les fichiers décrits.

9.1.4 Routage des requêtes dans une application Zend

Une requête HTTP adressée à l'application a le format suivant :

http://zscope.local/`ctrl`*/*`action`*[params]*

Ici, `ctrl` et `action` représentent respectivement le nom du contrôleur et le nom de l'action, et leur valeur par défaut est `index`. La forme de la requête est étrange puisqu'on semble faire référence à un sous-répertoire `ctrl` de la racine du site web, et même à un sous-répertoire `action`. Ces répertoires, comme nous l'avons vu, n'existent pas, mais le mécanisme de redirection renvoie la requête vers *index.php*. À ce moment-là un processus de *routage* implanté par le Zend Framework analyse la requête et détermine le contrôleur et l'action demandés. La méthode implantant l'`action` du contrôleur `ctrl` est alors exécutée.

Si, par exemple, on appelle *http://zscope.local/model/simpletbl*, la méthode `simpletblAction()` du contrôleur *ModelController* sera exécutée par le script *index.php*.

Cette réécriture permet de normaliser l'adressage des fonctionnalités d'une application Zend. Les paramètres passés à un action peuvent l'être soit sous la forme standard *?nom=valeur*, soit à nouveau sous la forme */nom/valeur*.

Il est temps d'inspecter le fichier *index.php*, nommé *bootstrap file*, ou *fichier d'amorçage*, qui se charge d'initialiser l'application et de déclencher le processus de routage.

Exemple 9.1 *zscope/www/index.php* : *Le* fichier d'amorçage *de l'application* ZScope

```
<?php
// On affiche toutes les erreurs
error_reporting(E_ALL | ~E_STRICT);

// On recherche le chemin d'accès à ZScope
$root = dirname(dirname(__FILE__)) . DIRECTORY_SEPARATOR ;

// On ajoute le chemin d'accès au ZF
set_include_path(dirname($root) . DIRECTORY_SEPARATOR . 'ZF'
. DIRECTORY_SEPARATOR . PATH_SEPARATOR . get_include_path()
);

// Ajout des chemins d'accès aux composants de l'application
set_include_path('.' .
PATH_SEPARATOR . $root . 'application' . DIRECTORY_SEPARATOR .
                 'models' . DIRECTORY_SEPARATOR .
PATH_SEPARATOR . get_include_path()
);

// On utilise toujours le loader automatique
require_once 'Zend/Loader.php';
Zend_Loader::registerAutoload();

// On lit le fichier de configuration
$config = new Zend_Config_Ini("../application/config.ini", "
    staging");
ini_set('display_errors', $config->app->display_errors);
date_default_timezone_set($config->app->default_timezone);

// Connexion à la base de données
$db = Zend_Db::factory($config->db);

// Cette connexion est à utiliser pour le Modèle
Zend_Db_Table::setDefaultAdapter($db);

// On ajoute la configuration et la connexion
// au registre de l'application
$registry = Zend_Registry::getInstance();
$registry->set('config', $config);
$registry->set('db', $db);

// Initialisation et exécution d'un contrôleur frontal
try {
  $front = Zend_Controller_Front::getInstance();
  // $front->throwExceptions(true);
  $front->setControllerDirectory('../application/controllers');
```

```
    // Utilisation du layout pour l'affichage
    Zend_Layout::startMvc();

    // C'est parti!
    $front->dispatch();
}
catch (Zend_Exception $e) {
    echo "Erreur dans l'initialisation du site.<br/>"
    . "<b>Message:</b> " . $e->getMessage()
    . " in " . $e->getFile() . "à la ligne " . $e->getLine() . "<br
        />";
}
```

Initialement, on n'y comprend rien, même si les commentaires disent assez clairement le rôle des différentes initialisations effectuées. Regardons dans le détail.

Un point important est l'ajout de répertoires dans la liste des chemins d'inclusion de PHP, afin d'accéder d'une part aux classes du Zend Framework, d'autre part aux classes de l'application elles-mêmes. On utilise le *loader* du ZF pour charger automatiquement les classes sans avoir besoin d'effectuer de nombreux appels à `require_once()`. Le chargement automatique s'appuie sur une convention largement suivie dans les bibliothèques PHP, consistant à établir une correspondance entre le nom d'une classe et le fichier qui contient sa définition. La classe `Zend_Db_Adapter` par exemple se trouve dans le fichier *Zend/Db/Adapter.php*. Comme nous avons placé */usr/local/share/ZF/library* dans nos chemins d'inclusion, et que *library* contient le répertoire *Zend* qui est la racine de tous les composants Zend, le *loader* trouve le fichier contenant la classe et la charge automatiquement dès qu'on en a besoin. Plus besoin d'inclusions explicites.

Quand l'environnement est initialisé, on peut effectuer le routage de la requête HTTP reçue. Il consiste essentiellement à analyser l'URL de la requête pour déterminer le contrôleur et l'action demandées, à charger la classe correspondant au contrôleur puis à réaliser l'action. Comme dans notre MVC, un contrôleur spécial, le *frontal*, se charge du routage. C'est un « singleton » (on ne peut instancier qu'un seul objet de la classe) et sa méthode principale est `dispatch()` qui distribue les requêtes aux contrôleurs.

```
// Initialisation et exécution d'un contrôleur frontal
$front = Zend_Controller_Front::getInstance();
$front->throwExceptions(true);
$front->setControllerDirectory('../application/controllers');

// Utilisation du layout pour l'affichage
Zend_Layout::startMvc();

// C'est parti!
$front->dispatch();
```

Par défaut, le ZF ne lève pas d'exception, mais sur un site de test comme le nôtre il est préférable, au moins initialement, d'afficher les exceptions avec la commande *throwExceptions(true)*.

Il reste des parties du fichier *index.php* non encore commentées : la lecture du fichier de configuration, la connexion à la base de données, leur placement dans le registre, et l'appel à *startMvc()*. Elles sont traitées dans les sections qui suivent.

9.1.5 Configuration

Toute application a besoin de fichiers de configuration dans lesquels on place des informations dépendantes d'un contexte particulier d'exécution, comme les paramètres de connexion à la base de données.

Une solution simple est d'utiliser des tableaux PHP auxquels on peut directement accéder dans un script. Le Zend Framework propose une option plus puissante, basée sur la syntaxe des fichiers INI (comme le *php.ini*). Cette option présente l'avantage d'un mécanisme d'héritage et de replacement de valeurs très pratique pour configurer des environnements proches les uns des autres.

Voici le fichier `config.ini`, que nous avons choisi de placer dans *application* (évidemment pas dans *www* !).

Exemple 9.2 *zscope/application/config.ini* : *Le fichier de configuration*

```
[production]
#
# Site configuration
#
app.name = "ZScope"
app.base_url =
app.display_errors = 0
app.admin_mail = philippe.rigaux@dauphine.fr
app.default_timezone = Europe/Paris
app.cookie.lifetime = 3

db.adapter = Pdo_Mysql
db.params.host = localhost
db.params.dbname = Films
db.params.username = adminFilms
db.params.password = mdpAdmin

[staging: production]
app.display_errors = 1
```

Le contenu du fichier est libre, mais sa structure doit obéir à quelques règles. Tout d'abord, il est décomposé en *sections*. Ici, nous en avons deux, `production` et `staging`, cette dernière définissant l'environnement de test de l'application. La syntaxe `[staging: production]` indique que `staging` *hérite* de

`production`. En d'autres termes, toutes les valeurs non redéfinies dans `staging` sont identiques à celles de `production`. Les autres sont redéfinies. Ainsi, la valeur de `display_errors` est à 1 pour `staging` et à 0 pour `production` : on affiche les erreurs dans un environnement de test, pas dans un environnement de production.

Ensuite, les valeurs du fichier de configuration sont organisées hiérarchiquement, en *domaines*. Nous avons ici deux domaines principaux, `app` et `db`, et vous êtes libres de définir vos domaines comme vous l'entendez. Un domaine peut lui-même avoir des sous-domaines, comme `db` qui dispose d'un sous-domaine `params`.

Le ZF charge un fichier de configuration dans un objet comme suit (ce qui mériterait d'être accompagnée d'un test en cas d'échec) :

```
$config = new Zend_Config_Ini("../application/config.ini", "
    staging");
```

On indique la section que l'on veut charger. Il suffit de changer `staging` en `production` pour passer de l'environnement de test à celui de production.

La structure de l'objet reflète la structuration en domaines de la configuration. Il existe donc un objet `config->app` pour le domaine `app`, un objet `$config->db` pour le domaine `db`, un objet `$config->db->params` pour le domaine `db.params`, etc. On accède à une valeur comme à une propriété d'un objet, par exemple `$config->app->display_errors` pour le choix d'afficher ou non les erreurs.

9.1.6 Connexion à la base de données

Le Zend Framework propose une couche d'abstraction pour accéder de manière transparente aux bases de données relationnelles, quel que soit le serveur. Une connexion est une instance de `Zend_Db_Adapter` créée par une « *factory* » (« usine » instanciant des objets en fonction du contexte). Il faut lui passer en paramètres deux informations :

1. le nom de l'adaptateur à utiliser (il existe un adaptateur pour quasiment tous les SGBD) ;
2. les paramètres usuels de connexion.

La méthode la plus simple consiste à définir ces paramètres dans le fichier de configuration (voir page 365) dans le domaine `db`. Une fois le fichier de configuration chargé, il suffit de passer l'objet `$config->db` à la *factory* pour obtenir la connexion

```
// Connexion à la base de données
$db = Zend_Db::factory($config->db);

// Cette connexion est à utiliser pour le Modèle
Zend_Db_Table::setDefaultAdapter($db);
```

Cette connexion peut être utilisée de manière classique pour effectuer des requêtes SQL. Elle peut également servir de support pour les composants du Zend Framework qui établissent une correspondance entre le Modèle et la base. L'appel à

`setDefaultAdapter()` indique que la connexion est utilisée par défaut dans ce cadre. Voir page 379 pour des exemples de recours à cette connexion.

Détail potentiellement important : la véritable connexion à la base ne s'effectue qu'avec la première requête effectuée, et c'est à ce moment-là qu'on risque de s'apercevoir que la connexion échoue à cause d'un problème de mot de passe ou autre. Il est possible d'appeler la méthode `getConnection()` sur l'objet `$db` pour forcer la connexion et vérifier qu'elle s'exécute correctement.

9.1.7 Le registre

Le *bootstrap file* stocke les objets `$config` et `$db` dans le *registre* de l'application Zend. Il s'agit d'un espace global dans lequel on peut placer des informations utiles partout dans l'application, ce qui évite d'avoir à les passer systématiquement en paramètre. C'est une solution plus propre que l'utilisation des variables globales. Ici, on place l'objet dans une entrée du registre nommée `config`. Nous verrons un peu plus loin comment accéder à ces entrées.

```
$registry = Zend_Registry::getInstance();
$registry->set('config', $config);
```

9.1.8 Contrôleurs, actions et vues

Regardons ce qui se passe quand on accède simplement à l'URL *http://zmax.local*. Le frontal détermine tout d'abord que ni le contrôleur, ne l'action ne sont spécifiés. Il prend donc ceux par défaut : `Index` pour le contrôleur et `index` pour l'action. Le contrôleur est implanté par une classe nommée `IndexController` dans le répertoire *application/controllers*, et chaque méthode de cette classe est une action nommée *nom*`Action`. Voici le code, très basique, de notre contrôleur par défaut;

Exemple 9.3 *zscope/application/controllers/IndexController.php* : *le contrôleur* `Index`.

```
<?php

class IndexController extends Zend_Controller_Action
{
  /**
   * L'action par défaut. Elle affiche juste la page
   * d'accueil.
   */

  function indexAction()
  {
    $this->view->titre_page = "Page d'accueil";
  }

  /**
```

```
     * Quelques informations sur la configuration
     */

    function configAction()
    {
        $this->view->titre_page = "Contenu du fichier de
            configuration";

        // On prend la configuration dans le registre, on la met dans
        // la vue
            $registry = Zend_Registry::getInstance();
            $this->view->config = $registry->get('config');;
    }
}
```

On remarque d'abord que tout contrôleur est une sous-classe de `Zend_Controller_Action`. Un contrôleur hérite donc des propriétés de cette super-classe, notamment de `$this->view` qui est le composant Vue avec lequel nous pouvons produire le document résultat. On peut placer des informations dans la vue avec une affectation `$this->view->`*`nom`* `= valeur`.

On peut se demander d'où vient le fragment HTML qui constitue la page d'accueil du site ZSCOPE, produite par cette action. Dans le Zend Framework, beaucoup de décisions s'appuient sur des conventions de nommage et d'emplacement, ce qui évite de se répéter. La convention pour les vues est la suivante : Le fragment (ou *script*) associé à une action nommée *`act`* du contrôleur *`ctrl`* se trouve dans le répertoire *views/scripts/*`ctrl`, et se nomme *`act.phtml`*. Ici, l'action `index` se conclut donc automatiquement par l'exécution du script *views/scripts/index/index.phtml*.

Vous pouvez consulter ce fichier. Il contient le code HTML qui s'affiche dans la zone principale de la page d'accueil de ZSCOPE. Tout le reste de la page, incluant le logo, le menu, les différents tableaux imbriqués spécifiant le positionnement des différentes parties et notamment de la zone principale d'affichage, relève du *layout*. La gestion de ce dernier sera présentée page 376.

Prenons un autre exemple avec l'action `configAction()` du contrôleur `Index` (voir ci-dessus). Le code est très simple : on récupère dans le registre l'objet de configuration et on l'affecte à la vue.

La vue, un script PHP un peu spécial, est alors exécutée. Au sein de ce script on dispose de l'objet `$config` que l'on vient d'affecter, ainsi que de la chaîne de caractères `titre_page`.

Exemple 9.4 *zscope/application/views/scripts/index/config.phtml* : *Le script Vue de l'action* *`config`* *du contrôleur* *`Index`*

```
Voici quelques informations extraites du fichier de configuration

<h4>Informations générales sur l'application</h4>
```

```
<ol>
<li>Nom de l'application: <?php  echo $this->config->app->name;
    ?></li>
 <li>Affichage des erreurs: <?php  echo $this->config->app->
     display_errors; ?></li>
<li>Administrateur: <?php  echo $this->config->app->admin_mail;
    ?></li>
</ol>

<h4>Connexion à la base de données</h4>

<ol>
<li>Pilote PDO: <?php  echo $this->config->db->adapter; ?></li>
 <li>Serveur: <?php  echo $this->config->db->params->host; ?></li
     >
  <li>Base: <?php  echo $this->config->db->params->dbname; ?></li
      >
   <li>Login: <?php  echo $this->config->db->params->username;
       ?></li>
</ol>
```

On programme dans la vue comme dans n'importe quel script PHP, avec la particularité que les données « dynamiques » sont disponibles dans l'objet `$this` parce qu'elles ont été placées là par l'action associée à la vue. Le script-vue ne sert qu'à mettre en forme ces données, pas à les créer ou les modifier. Nous reviendrons sur la gestion de la vue dans le Zend Framework page 376.

Le moment est probablement venu pour vous de faire une pause. Parcourez l'arborescence des fichiers, et modifiez ceux qui interviennent dans l'exécution de l'action `index` du contrôleur `Index`. Il s'agit de *controllers/IndexControlller.php* et *views/script/index/index.phtml*. Clarifiez les interactions entre ces compoants (contrôleur, action et vue : nous n'avons pas encore vu le modèle). La suite revient plus longuement sur les composants Zend relatifs au MVC et aux bases de données.

9.2 ACCÈS À LA BASE DE DONNÉES

Le contrôleur `bd` de l'application ZSCOPE montre comment utiliser la connexion à la base, instance de `Zend_Db_Adapter`, pour accéder classiquement à une base de données par des requêtes SQL. Sur ce point, le ZF s'appuie très largement sur PDO (voir page 238) qui propose déjà tout ce qui est nécessaire à un accès normalisé aux bases relationnelles. Ce qui suit n'est donc qu'une introduction assez rapide, que vous pouvez compléter avec la documentation PDO en ligne.

9.2.1 Interrogation

La méthode `query()` prend une requête et retourne une instance de `Zend_Db_Statement` qui permet de parcourir le résultat. Le mécanisme est totalement standard et conforme à ce que nous utilisons depuis le début du livre. Voici l'action `queryfetch()` qui interroge la table *Film*.

```
function queryfetchAction ()
{
  // On récupère la connexion dans le registre
  $bd = Zend_Registry::getInstance()->get('db');

  // Exécution de la requête
  $result =
     $bd->query("SELECT id, titre, annee FROM Film WHERE annee
         > 1980");

  // Itération sur les films
  $this->view->films = array();
  while ($film = $result->fetch (Zend_Db::FETCH_OBJ)) {
    $this->view->films[] = $film;
  }
 }
```

Essentiellement, on interroge la base, puis on itère sur le résultat en plaçant chaque objet dans un tableau de `$this->view`. Le paramètre de la méthode `fetch()` indique le format dans lequel on souhaite récupérer chaque ligne. Comme d'habitude, au lieu d'`OBJ`, on peut utiliser `ASSOC` pour des tableaux associatifs, `NUM` pour des tableaux indicés, et quelques autres options moins utiles. Le script de vue parcourt ce tableau et crée le document HTML de présentation. Notez la méthode `escape()` de la vue qui s'assure que les caractères perturbant le document HTML sont neutralisés.

Exemple 9.5 *zscope/application/views/scripts/bd/queryfetch.phtml* : *La vue affichant la liste des films*

```
Voici la liste des films trouvés dans la base.

<ol>
<?php foreach ($this->films as $film) { ?>

 <li><?php echo $this->escape($film->titre) ; ?></li>

<?php } ?>
</ol>
```

En général, on utilise des requêtes contenant des paramètres venant de formulaires ou autres, qu'il faut prendre garde d'échapper pour éviter les caractères posant problème. Les méthodes `quoteInto()` et `quote()` effectuent cet échappement en tenant compte des règles propres au serveur auquel on est connecté. Ces régles

varient d'un SGBD à un autre. Pour MySQL, nous savons qu'il faut placer une barre oblique inversée devant les « ' », pour ORACLE il faut les doubler, etc. Voici un exemple avec `quoteInto()`.

```
function quoteAction ()
{
 // On récupère la connexion dans le registre
  $bd = Zend_Registry::getInstance()->get('db');

  // Le titre du film cherché
  $titre = "Jeanne d'Arc";
  $this->view->titre = $titre;

  // Préparation de la requête
  $requete = $bd->quoteInto("SELECT * FROM Film WHERE titre=?",
       $titre);
  $result = $bd->query($requete);

  // Itération sur les films
  $this->view->films = array();
  while ($film = $result->fetch (Zend_Db::FETCH_OBJ)) {
    $this->view->films[] = $film;
  }
 }
```

Il existe tout un catalogue de méthodes `fetch` pour récupérer, au choix, toutes les lignes d'un résultat, ou bien sa première ligne, ou bien la première colonne de sa première ligne, etc. Voici l'action `fetch` qui vous donnera une idée de l'utilisation de ces méthodes. La documentation de Zend sur le composant `Zend_Db` détaille la liste de toutes les possibilités.

```
/**
  * Quelques méthodes fetch*** def Zmax_Db_Adapter
  */
 function fetchAction ()
 {
 // On récupère la connexion dans le registre
   $bd = Zend_Registry::getInstance()->get('db');

   // Tous les films contenant un 'v',
  $requete = "SELECT titre, annee, genre FROM Film WHERE titre
      LIKE '%v%'";
  $this->view->requete = $requete;

   // Attention, on veut toujours récupérer des objets
  $bd->setFetchMode(Zend_Db::FETCH_OBJ);

  // Fetch: comme un tableau d'objets
   $this->view->films = $bd->fetchAll($requete);

   // On prend une seule ligne, la première du résultat
   $this->view->film = $bd->fetchRow($requete);
```

```
    // On prend une valeur dans une ligne
    $this->view->titre = $bd->fetchOne($requete);
}
```

Avec ces méthodes `fetch`, l'exécution de la requête et la récupération du résultat s'effectuent en un seul appel. Il est souvent pratique de pouvoir récupérer facilement une ligne ou une valeur spécifique dans la base de données.

9.2.2 Insertion et mise à jour

Les modifications de la base, que ce soit insertion ou mises à jour, bénéficient également de méthodes génériques. Ces méthodes effectuent automatiquement les échappements nécessaires pour s'assurer que les modifications s'exécutent sans problème. Voici l'action `insert` illustrant une insertion.

```
/**
 * Comment insérer dans une table
 */

function insertAction()
{
 // On récupère la connexion dans le registre
  $bd = Zend_Registry::getInstance()->get('db');

  // On veut insérer l'Espagne. On commence par supprimer
  // si elle existe déjà.
  $bd->query ("DELETE FROM Pays WHERE code='ES'");

  // Maintenant on définit le tableau des données à insérer
  $espagne = array ("code" => "ES", "nom" => "Espagne", "langue
      " => "Espagnol");

  // On insère et on récupère le nombre de lignes insérées
  $this->view->nb_lignes = $bd->insert("Pays", $espagne);
}
```

Pour être complet, voici le code d'une mise à jour.

```
function updateAction()
{
 // On récupère la connexion dans le registre
  $bd = Zend_Registry::getInstance()->get('db');

  // Tableau des colonnes à modifier
  $update_val = array ("langue" => "Anglais US");

  // Clause where pour la modification
  $where[] = "code='US'";

  $nb_rows = $bd->update("Pays", $update_val, $where);
  }
```

9.3 LE MVC DU ZEND FRAMEWORK

Nous avons déjà vu l'essentiel de l'organisation du Zend Framework pour les contrôleurs et les actions[2]. Voici quelques compléments d'utilisation courante. Le code décrit dans cette section appartient au contrôleur *MvcController.php*.

9.3.1 L'objet `request`

Les paramètres reçus par une action sont placés dans un objet `request` que l'on obtient avec la méthode `getRequest()` d'un contrôleur. De nombreuses méthodes applicables à cet objet servent à récupérer les informations relatives à la requête HTTP qui a déclenché l'action. En voici une liste non exhaustive.

1. `getControllerName()`
2. `getActionName()`
3. `getMethod()`
4. *getQuery($key=null, $defaut=null)*
5. *getPost($key=null, $defaut=null)*
6. *getCookie($key=null, $defaut=null)*
7. *getParam($key=null, $defaut=null)*

À chaque fois si `$key` est `null`, on obtient un tableau donnant toutes les informations disponibles (par exemples, tous les *cookies*). L'exemple qui suit montre comment récupérer tous les paramètres passés à une action. Notez également que `request` sert à obtenir le nom du contrôleur et de l'action courante.

```
function requestAction()
 {
   // Récupérons la requête
   $request = $this->getRequest();

   // On obtient le nom du contrôleur et de l'action
   $this->view->nom_controleur = $request->getControllerName();
   $this->view->nom_action = $request->getActionName();

   // Prenons les paramètres HTTP
   $params = $request->getParams();
   $this->view->liste_params = "";
   foreach ($params as $nom => $valeur)
      $this->view->liste_params .= " ($nom = $valeur) ";
 }
```

Au lieu de parler de paramètres HTTP, il serait plus judicieux de parler de paramètres en général, transmis à une méthode dans une classe orienté-objet. Il est possible en effet, avec le Zend Framework, de déclencher une action à partir

2. Un troisième niveau dans la hiérarchie, les *modules*, est possible.

d'une autre action (avec un `forward()`) en lui passant un objet `request` créé de toutes pièces sans passer par HTTP. Le mécanisme d'encapsulation obtenu par la programmation objet a pour effet de fournir des applications et des composants qui ne sont plus liés à un environnement d'exécution particulier.

9.3.2 L'objet `response`

Le complément de l'objet `request` est l'objet `response` qui encapsule les méthodes gérant la production du résultat (ceux qui ont déjà programmé en Java/Servet/JSP ne seront pas dépaysés). L'objet `response` est utile par exemple quand on produit un résultat non HTML, auquel cas il faut modifier les entêtes de la réponse HTTP, sans recourir au composant Vue.

L'exemple suivant montre comment renvoyer un document en texte brut.

```
function responseAction()
{
  // Ne pas utiliser la vue
  $this->getHelper('ViewRenderer')->setNoRender();

  // Modifier l'en-tête HTTP donnant le type du document
  // renvoyé
  $this->getResponse()->setHeader('Content-type', 'text/plain')
      ;

  $reponse = "Je suis une réponse en texte pur";
  $this->getResponse()->setBody($reponse);
}
```

9.3.3 Gérer les exceptions

Pour conclure cet aperçu des techiques de programmation MVC avec le Zend Framework, revenons sur la gestion des exceptions. Pour l'instant, nous avons demandé au contrôleur frontal de lever des exceptions avec l'instruction suivante placée dans le fichier *index.php*.

```
$front->throwExceptions(true);
```

Ce n'est pas vraiment une bonne méthode car une exception sera redirigée vers le bloc `catch` du fichier *index.php* qui va devoir tout gérer. La gestion par défaut des exceptions consiste à déclencher une action `error` d'un contrôleur *ErrorController.php*.

Ce contrôleur existe dans ZSCOPE. Commentez l'appel `throwException()` dans *index.php*, et déclenchez une exception pour voir ce qui se passe. Vous pouvez tout simplement déclencher l'action `exception` du contrôleur `mvc`, dont voici le code.

```
 function exceptionAction()
{
  throw new Zend_Exception ("J'envoie une exception");
}
```

Cette action lève une exception, dirigée vers le contrôleur `Error`. La requête créée pour cette redirection contient un paramètre `error_handler` qui analyse l'exception levée. Cette dernière peut de deux types :

1. soit une erreur déclenchée par le Zend Framework lui-même, par exemple un contrôleur ou une action qui n'existe pas ;
2. soit une exception levée par l'application.

Voici le code complet de l'action `error` du contrôleur `Error` pour l'application ZSCOPE. On produit un message indiquant de manière complète l'endroit où l'exception a été levée, ainsi que le message associé.

```
function errorAction()
{
  $this->titre_page = "Une exception a été rencontrée";

  // On récupère le gestionnaire d'exception
  $eh = $this->_getParam('error_handler');

  if (is_object($eh)) {
    $errmess =  $script = $line  = "";
    $context = "Erreur";

    // Vérifions le type d'exception rencontrée
    switch ($eh->type) {
      case Zend_Controller_Plugin_ErrorHandler::
          EXCEPTION_NO_CONTROLLER:
      case Zend_Controller_Plugin_ErrorHandler::
          EXCEPTION_NO_ACTION:
        $context = "Erreur Zend MVC ";
        $errmess = "Contrôleur ou action inconnue";
        break;

      case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_OTHER:
      default:
        $exception = $eh->exception;
        $script = $exception->getFile();
        $line = $exception->getLine();
        $errmess = $exception->getMessage();
        $context = get_class($exception);
        break;
    }
    // On crée le message
    $this->view->message =
       "($context) Script $script ligne $line: $errmess";
  }
  else {
    $this->view->message = "Erreur interne";
  }
}
```

Cette action devrait encore être améliorée pour tenir compte du contexte. Dans une application en production il faudrait afficher un message neutre sur l'indisponibilité de l'application, et envoyer un em-ail à l'administrateur du site avec le message précis de l'erreur rencontrée.

9.4 LA VUE DANS LE ZEND FRAMEWORK

Cette section revient sur le système de vues proposé par le Zend Framework.

9.4.1 Les vues sont des scripts PHP

Un script de vue a pour objectif de produire un document. La solution proposée par Zend pour gérer les vues dans l'architecture MVC a le mérite de la simplicité. Le langage de templates est PHP lui-même ; la seule particularité est que la vue dispose d'un ensemble de données à afficher qui lui ont été affectées par l'action associée.

L'exemple suivant est la vue *fetch.phtml* associée à l'action `fetch` du contrôleur `Bd`, présenté un peu plus haut. Cette vue a reçu trois types de données : un tableau d'objets (des films), un objet (un film) et une chaîne de caractères (un titre).

Exemple 9.6 *zscope/application/views/scripts/bd/fetch.phtml* : *la vue de l'action* `fetch`

```
Voici quelques exemples de méthodes <tt>fetch</tt>
pour la requête
"<i><?php echo $this->escape($this->requete) ; ?></i>".

<h2>Méthode <tt>fetchAll</tt></h2>

<ol>
<?php foreach ($this->films as $film) { ?>

 <li><?php echo $this->escape($film->titre) ; ?></li>

<?php } ?>
</ol>

<h2>Méthode <tt>fetchRow</tt></h2>

On ne trouve qu'une ligne avec le film
<i><?php echo $this->escape($this->film->titre)
    . " paru en " . $this->film->annee; ?></i>.

<h2>Méthode <tt>fetchOne</tt></h2>

On ne prend que la première valeur de la première ligne.
Ici le titre <i><?php echo $this->escape($this->titre);?></i>.
```

La vue agit comme un script PHP, avec toute la puissance potentielle du langage. D'un côté, cela résout très simplement le problème des boucles ou des tests dans

des templates, qui mène à des langages de templates parfois inutilement compliqués. D'un autre côté, on se retrouve avec des scripts qui ouvrent et ferment sans cesse les balises PHP, ce qui peut devenir lassant et donne un résultat moins facile à lire.

Attention également à ne pas utiliser un script-vue pour faire de la programmation PHP. C'est possible avec cette approche, et il faut bien garder en tête que la vue ne doit implanter aucune intelligence, aucune logique. Elle doit juste produire un document.

Zend permet le remplacement de son système de vues par un autre, basé par exemple sur des templates. Il faut implanter l'interface `Zend_View_Interface`. Un exemple est donné dans la documentation pour les templates `Smarty`.

9.4.2 Le *layout*

Quand on développe un site, toutes les pages ont une présentation commune, désignée par le terme de *layout*. Si le système de vues Zend se limitait à un script pour chaque action, cela rendrait assez difficile, ou peu élégant, la gestion de ce *layout* commun. On pourrait produire des ent-êtes et des pieds de page, mais cette solution laborieuse rend plus difficile la conception du *layout* comme un document HTML homogène et bien formé.

C'est ici qu'intervient l'instruction `startMvc()` placée dans *index.php*, que nous n'avions pas encore commentée. Le *layout* implante la charte graphique du site. C'est un document nommé *layout.phtml*, situé dans *views/scripts*, qui consiste essentiellement en un document HTML, avec quelques instructions comme celle-ci :

```
<?php echo $this->escape($this->titre_page); ?>
```

Tout se passe comme si le *layout* était un script PHP standard dans lequel on peut insérer des instructions `echo`. Les données disponibles pour l'affichage sont les propriétés d'un objet `$this` qui constitue le contexte du *layout*. Le *layout* contient également une instruction assez spéciale.

```
<?php echo $this->layout()->content; ?>
```

De quoi s'agit-il ? Le système de vues proposé par défaut par le ZF fonctionne à deux niveaux. Quand une action s'exécute, le document résultat, obtenu par exécution du template associé à l'action, est placé dans une entité *content*. Cette entité peut alors, dans un second temps, être insérée à l'endroit approprié dans le *layout* qui constitue le second niveau.

Pour prendre l'exemple de notre action `index` dans le contrôleur `Index`, le template *index.phtml* est tout d'abord exécuté, et donne pour résultat un fragment HTML. Ce fragment est affecté à l'entité `content` du *layout*. Le *layout* lui-même est finalement affiché.

> **REMARQUE –** C'est l'instruction `startMvc()` placée dans le *boostrap file* qui établit cette réalisation à deux niveaux de la vue. On peut omettre cette instruction et se contenter d'un seul niveau (essayez…), mais il devient alors assez difficile d'organiser avec logique les fichiers constituant la vue.

On peut gérer le *layout* au niveau de chaque action, soit pour modifier le document utilisé avec l'instruction

```
// Utiliser autre.phtml comme layout
$this->_helper->layout()->setLayout('autre');
```

soit en n'utilisant plus du tout le layout, par exemple pour produire un document non écrit en HTML.

```
// Ne pas utiliser le layout
$this->_helper->layout()->disableLayout();
```

9.4.3 Créer des *Helpers*

La notion de *Helper* (assistant en français) correspond à une manière détournée d'enrichir une classe orientée-objet sans recourir à l'héritage. Nous allons prendre le cas d'un *helper* pour la vue. Le point de départ est le suivant : si on veut enrichir la vue avec des méthodes utiles pour tout le site, comme la mise en forme d'une date, ou d'un montant monétaire, la solution naturelle est de créer une sous-classe `MaVue` de `Zend_View` et d'y placer ces méthodes.

Pour éviter cette démarche un peu lourde, le Zend Framework permet d'implanter les méthodes ajoutées sous forme de *helper* dont le nom et la syntaxe particulière mènent à les traiter comme des méthodes de la `Zend_View`. Voici un exemple simple. On veut pouvoir disposer, dans chaque vue, de l'URL de base du site. Cette URL est vide si le site est directement dans la racine *htdocs* du site web, sinon elle doit contenir le chemin d'accès entre *htdocs* et la racine du site.

Pour constituer correctement les URL placées dans une vue, il faut les préfixer par l'URL de base. Il serait dangereux de la placer « en dur » dans les vues, sous peine d'avoir à changer beaucoup de choses si l'organisation du serveur évolue. On doit donc disposer de cette URL de base dans les scripts de vue.

Pour cela, on place dans le répertoire *views/helpers* le code suivant :

Exemple 9.7 *zscope/application/views/helpers/BaseURL.php* : *la méthode ajoutée à la vue pour obtenir l'URL de base du site.*

```
<?php
/**
 *
 * Exemple d'un 'helper' pour la vue, donnant la
 * base de l'application.
 *
 */

class Zend_View_Helper_BaseUrl
{
  /**
   * On prend simplement l'URL de base dans la configuration
   *
   */
```

```
  function baseUrl()
  {
    $registry = Zend_Registry::getInstance();
          $config = $registry->get('config');;
    return $config->app->base_url;
  }
}
```

Il s'agit s'une classe dont le nom est préfixé par `Zend_View_Helper`. La partie variable du nom de la classe indique le nom de la méthode étendant la `Zend_View`, soit, ici, `baseUrl()`.

On peut utiliser cette méthode comme si elle appartenait à la vue. Voici par exemple comment on insère dans le *layout* le logo du site en préfixant par l'URL de base le chemin d'accès au répertoire *images* :

```
<img src="<?php echo $this->BaseUrl();?>/images/logo.png"
                        border="0" alt="Webscope" title="Accueil"
                            />
```

La vue Zend est déjà pré-équipée avec de nombreux *helpers* servant, par exemple, à faciliter la production de formulaires. Je vous renvoie à la documentation pour plus de détails à ce sujet.

9.5 LE COMPOSANT *MODÈLE* DU ZEND FRAMEWORK

La technique dite *Object-Relational Mapping* (ORM) associe une couche orientée-objet à une base relationnelle sous-jacente. Elle est très importante pour réaliser le Modèle d'une application, puisque la plupart des objets du modèles sont persistants et doivent être sauvegardés dans la base de données.

9.5.1 L'ORM du Zend Framework

Zend propose un ORM pour établir la correspondance entre la base et le code PHP. Cet ORM est moins puissant (du moins à l'heure où ces lignes sont écrites) que d'autres comme *Ruby On Rails* ou *Hibernate*, mais il gagne en simplicité et peut-être en performances. Cela constitue une manière élégante de cacher la structure relationnelle et de s'épargner dans de nombreux cas (mais pas *tous*) l'expression de requêtes SQL répétitives.

L'ORM du ZF ne s'appuie pas sur de longs documents de configuration XML. Il se contente de réclamer quelques définitions légères dans des classes dédiées, et fournit ensuite des mécanismes sympathiques de parcours dans la base de données en suivant les liens définis par les clés primaires et étrangères. Toutes les classes ORM d'une application doivent hériter de l'une des trois classes abstraites suivantes.

1. `Zmax_Db_Table_Abstract` assure la correspondance avec les tables de la base ;

2. `Zmax_Db_Rowset_Abstract` eassure la correspondance avec les ensembles de lignes (autrement dit, les résultats de requêtes) ;
3. `Zmax_Db_Row_Abstract` assure la correspondance avec les lignes d'une table ou d'un résultat de requête.

Comme leur nom l'indique, ces classes sont *abstraites* ; on ne peut donc pas les instancier directement. Il faut fournir une classe concrète, héritant de `Zmax_Db_Table_Abstract`, pour chaque table intervenant dans le modèle ORM de l'application.

9.5.2 Le modèle ORM de l'application

Le premier exemple est la classe `Artiste`, assez simple à définir puisqu'elle n'a pas de clé étrangère la liant à une autre table dans notre modèle de données. La correspondance est définie dans un fichier *Artiste.php* situé dans *application/models*.

Exemple 9.8 *zscope/application/models/Artiste.php* : *Le modèle de la table Artiste*

```
<?php

class Artiste extends Zend_Db_Table_Abstract
{
  protected $_name = 'Artiste';
  protected $_primary = 'id';

  // Pas d'auto-incrémentation
  protected $_sequence = false;
}
```

Cet exemple représente la correspondance minimale d'une classe sous-typant `Zmax_Db_Table_Abstract`. Les propriétés suivantes doivent être définies :

1. `$_name` est le nom de la table ;
2. `$_primary` est le nom de la clé primaire (un tableau s'il y a plusieurs attributs) ;
3. `$_sequence` est à `true` (par défaut) si la clé est auto-incrémentée ; rappelons que pour des raisons de portabilité aucune de nos clés n'est auto-incrémentée, mais vous pouvez omettre cette propriété et lui laisser sa valeur par défaut si vous choisissez l'auto-incrémentation.

La clé primaire peut être obtenue directement de la base si elle n'est pas précisée (comme nous l'avons fait dans la classe `TableBD` implantant le modèle de notre MVC *light*). La faire figurer explicitement facilite la compréhension de la classe ORM.

Voyons maintenant un exemple pour la table *Film*, dans laquelle figurent deux clés étrangères, l'une vers la table *Artiste* (le metteur en scène) et l'autre pour le pays. Pour simplifier, nous ne donnons la correspondance ORM que pour la première.

Exemple 9.9 *zscope/application/models/Film.php* : *le modèle de la table Film*

```
<?php

class Film extends Zend_Db_Table_Abstract
{
  protected $_name = 'Film';
  protected $_primary = 'id';

  // Pas d'auto-incrémentation de la clé
  protected $_sequence = false;

  // Référence à la table Artiste
  protected $_referenceMap = array (
    "Realisateur" => array (    // Le "rôle" de l'association
        "columns" => 'id_realisateur', // La clé étrangère
        "refTableClass" => "Artiste", // La classe ORM de la
            table référencée
        "refColumns" => "id", // La clé étrangère référencée
    )
  );
}
```

Le tableau `$_referenceMap` contient un élément pour chaque table référencée par une clé étrangère. Au niveau de l'ORM, on ne donne pas la référence à des *tables*, mais aux classes ORM qui les encapsulent. Ici, on fait référence à la classe `Film` définie précédemment. Le but est bien d'avoir un modèle d'objets se référençant les uns les autres, en dissimulant les accès à la base de données.

Chaque élément de `$_referenceMap` est lui-même un tableau donnant l'information définissant le lien entre les deux tables sous-jacentes. Elle doit permettre la reconstitution de la jointure pour calculer le lien entre les lignes des deux tables. On retrouve e toutes les informations de la clause `FOREIGN KEY` en SQL.

Comme `$_referenceMap` est un tableau associatif, on donne un nom à l'association qui n'est pas nécessairement le nom de la table référencée, mais désigne plutôt le rôle de cette dernière dans l'association. Ici, l'entité référencée dans la table *Artiste* est le réalisateur du film, et ce rôle apparaît explicitement comme nom de l'association. On peut trouver plusieurs clés étrangères vers la même table, qu'il faut distinguer par un nom spécifique.

Le dernier exemple que nous donnerons montre la correspondance pour une association plusieurs à plusieurs. C'est la table *Role* qui établit cette association entre les films et leurs acteurs dans la base de données. Au niveau des classes ORM, nous avons besoin d'une classe `Role` définie comme suit :

Exemple 9.10 *zscope/application/models/Role.php* : *le modèle de la table Role*

```
<?php

class Role extends Zend_Db_Table_Abstract
```

```
{
  protected $_name = 'Role';
  protected $_primary = array('id_film', 'id_acteur');

  // Pas d'auto-incrémentation
  protected $_sequence = false;

  // Référence au film et à l'artiste
  protected $_referenceMap = array (
    "Film" => array (
        "columns" => 'id_film', // Nom de la clé étrangère
        "refTableClass" => "Film", // Classe de la table
            référencée
        "refColumns" => "id" // Clé primaire référencée
     ),
    "Artiste" => array (
        "columns" => 'id_acteur',  // Nom de la clé étrangère
        "refTableClass" => "Artiste", // Classe de la table
            référencée
        "refColumns" => "id" // Clé primaire référencée
     ),

   );
}
```

On met en œuvre les mêmes principes que précédemment avec le tableau `$_referenceMap`, qui contient cette fois deux entrées.

Une nouvelle application doit non seulement définir le schéma relationnel avec SQL, mais également le modèle des classes ORM. Il ne semble pas exister à l'heure actuelle d'outil pour engendrer automatiquement les classes à partir des tables, mais cela viendra sûrement. Soulignons que les classes ORM héritent d'un ensemble de fonctionnalités pour gérer les échanges avec la base de données, mais qu'elles constituent aussi une place de choix pour implanter la « logique métier » de l'application. Dans notre cas cette logique est limitée car notre application est essentiellement orientée vers l'exploitation d'une base de données, sans traitement complexe. En général, le modèle comprend des méthodes déterminant le comportement fonctionnel, « métier », des objets, en plus de leur caractère persistant.

Dernière remarque avant de passer à l'exploitation des fonctionnalités de persistance : ces classes sont des classes concrètes qui peuvent être instanciées en objets qui communiquent avec les tables de la base. Elles ont donc besoin d'une connexion. Celle-ci n'apparaît pas ici, car elle est définie une fois pour toutes par l'appel à `setDefaultAdapter()` dans le fichier *index.php*.

9.5.3 Manipulation des données avec les classes ORM

Les méthodes suivantes sont héritées par les classes ORM :

1. `find()` recherche des lignes en fonction d'une ou plusieurs valeurs de clés primaires et retourne un ensemble de lignes, instance de `Zend_Db_Rowset` ;
2. `fetchRow()` ramène une seule ligne, instance de `Zend_Db_Row` ;
3. `fetchAll()` effectue une recherche générale (il est possible de définir une clause `where` au préalable) et renvoie un ensemble de lignes ;
4. `info()` renvoie un tableau PHP donnant les informations connues sur le schéma de la table ;
5. `createRow()` crée une ligne ;
6. `insert()` insère une ligne dans la base ;
7. `update()` effectue une mise à jour.

Regardons quelques exemples, tous extraits du contrôleur `Model` de ZSCOPE. Le premier cherche un film d'identifiant 1.

```
function simpletbleAction ()
{
  $tbl_film  = new  Film ();
  $this->view->film = $tbl_film->find(1)->current();
}
```

D'abord, on instancie la classe ORM, et on obtient un objet qui fournit une interface orientée-objet avec le contenu de la table relationnelle. Ensuite, on appelle la méthode `find()` qui prend en entrée une valeur de clé primaire (ou un tableau de valeurs) et retourne un *rowset* que l'on peut parcourir par itération avec des méthodes comme `next()`, `previous()` et `current()`. L'exemple ci-dessus n'est d'ailleurs pas très robuste car il ne teste pas ce qui se passerait si aucune ligne n'était ramenée.

Quand une ligne instance de `Zmax_Db_Row_Abstract` est obtenue, on accède à ses propriétés de manière naturelle avec la syntaxe `$obj->nom`. L'aspect le plus intéressant des classes ORM est cependant la possibilité de « naviguer » vers les lignes de la base associées à la ligne courante. L'exemple suivant montre comment, depuis un artiste (pour simplifier, on a mis « en dur » son identifiant), on obtient tous les films mis en scène par cet artiste.

```
 function dependentAction ()
{
  // On prend Quentin Tarantino (d'identifiant 37)
  $tbl_artiste = new  Artiste();
  $artiste = $tbl_artiste->find(37)->current();

  // Maintenant cherchons les films mis en scène
  $films = $artiste->findFilm ();
  // Equiv à $artiste->findDependentRowset ("Film");

  // Et on stocke dans la vue
```

```
    $this->view->artiste = $artiste;
                $this->view->films = $films;
}
```

Il y a là comme un (petit) tour de magie. La méthode `findFilm()` n'existe pas dans la classe `Artiste.php`. L'appel est intercepté grâce à la méthode magique `__call()`, et l'ORM comprend que nous faisons référence à l'association liant un film et son metteur en scène. Ce lien est d'ailleurs défini dans `Film.php`, pas dans `Artiste`. L'ORM de Zend est assez souple pour reconstituer l'information en allant consulter la définition de la classe distante. Une autre syntaxe possible est

```
$artiste->$comp->findDependentRowset("Film").
```

Cet appel renvoie un ensemble de lignes, que l'on peut ensuite parcourir avec une boucle `foreach`, comme le montre la vue associée à cette action.

Exemple 9.11 *zscope/application/views/scripts/model/dependent.phtml* : *la vue associée à l'action* `dependent`

```
Nous avons trouvé le metteur en scène
<b>
<?php echo $this->artiste->prenom . " " .$this->artiste->nom ; ?>
</b>
<p>
Voici la liste des films qu'il a réalisés:
</p>

<ol>
<?php foreach($this->films as $film): ?>
<li>Films : <?php echo $film->titre; ?></li>
<?php endforeach; ?>
</ol>
```

Bien entendu, une association peut être parcourue dans les deux sens, comme le montre l'exemple suivant où, pour un film donné, on cherche son réalisateur.

```
function referenceAction ()
{
    // On prend un film (Vertigo)
    $tbl_film  = new Film();
    $film = $tbl_film ->find(1)->current();

    // On cherche le "parent"
    $artiste = $film->findParentArtiste ();
    // ou bien: findParentRow ("Artiste");

    // Puis on le place dans la vue
    $this->view->film = $film;
    $this->view->realisateur = $artiste;
}
```

Finalement, le dernier exemple montre comment suivre les associations plusieurs à plusieurs. La syntaxe de l'appel est

find<`classeDistante`>*Via*<`classeIntermédiaire`>,

où `classeIntermédiaire` est la classe qui implante l'association, et `classeDistante` la classe située à l'autre bout de l'association impliquant l'objet courant. Concrètement, voici comment à partir d'un film on obtient tous ses acteurs.

```
function manytomanyAction ()
 {
   $tbl_film  = new  Film();
   // On prend le film 61 (c'est 'Kill Bill')
   $film = $tbl_film->find(61)->current();

   // Chercher tous les acteurs en passant par Role
   // Equiv à $film->findManyToManyRowset ("Artiste", "Role");
   $acteurs = $film->findArtisteViaRole ();

   // On place dans la vue
   $this->view->film = $film;
   $this->view->acteurs = $acteurs;
 }
```

En résumé, le modèle ORM évite dans beaucoup de cas l'alternance entre la programmation PHP et la programmation SQL et mène à un code plus concis et plus lisible. La navigation d'une ligne à l'autre en suivant des associations correspond à une grande part des requêtes SQL, et l'ORM vient donc harmoniser en « tout objet » ces mécanismes.

9.6 POUR CONCLURE ...

Nous arrêtons là ce premier tour d'horizon des possibilités du *Zend Framework*. Il montre quelques aspects essentiels, et reste malgré tout limité à une petite partie des composants proposés par le ZF, qui ambitionne de couvrir à peu près tous les aspects envisageables de la programmation d'applications PHP.

L'apprentissage d'un *framework* suppose un certain investissement en temps et en efforts de compréhension. Ces efforts sont particulièrement justifiés pour des projets conséquents impliquant des équipes de plusieurs personnes. La documentation du ZF est bien faite et on trouve de plus en plus de tutoriaux sur le Web qui permettent de comprendre en profondeur tel ou tel aspect. J'espère que l'introduction qui précède vous aura mis le pied à l'étrier pour continuer dans de bonnes conditions.

10

Récapitulatif SQL

Nous présentons, dans ce chapitre, un récapitulatif des commandes SQL découvertes au fur et à mesure dans les chapitres précédents, ainsi que de nombreux compléments sur la syntaxe du langage. Depuis sa version 4.1, MySQL propose une version de SQL très complète et conforme à la norme SQL ANSI.

Bien entendu, nous prenons comme exemple la base de données *Films* qui devrait maintenant vous être familière. Pour faciliter l'apprentissage et éviter de trop manipuler des identifiants abstraits, les exemples sont construits sur une base où les films sont identifiés par leur titre. Celui-ci sert donc également de clé étrangère. Vous pouvez vous entraîner à SQL directement sur notre site, avec une fonction ajoutée au site de démonstration WEBSCOPE qui permet d'exprimer directement des requêtes SQL sur la base de données.

Au cours de ce chapitre, nous illustrerons nos exemples avec l'échantillon de la base *Films* présenté dans la figure 10.1. Notez que cette base est incomplète. Il n'y a pas d'acteur par exemple pour *Kagemusha*, et l'année de naissance de Jacques Dutronc manque. Ces absences serviront à illustrer certains aspects de SQL. Nous avons également supprimé certains attributs pour plus de clarté.

titre	année	id_realisateur	genre
Impitoyable	1992	20	Western
Van Gogh	1990	29	Drame
Kagemusha	1980	68	Drame
Les pleins pouvoirs	1997	20	Policier

Film

id	nom	prénom	année_naissance
20	Eastwood	Clint	1930
21	Hackman	Gene	1930
29	Pialat	Maurice	1925
30	Dutronc	Jacques	
68	Kurosawa	Akira	1910

Artiste

titre	id_acteur	nom_rôle
Impitoyable	20	William Munny
Impitoyable	21	Little Bill Dagget
Van Gogh	30	Van Gogh
Les pleins pouvoirs	21	Le président

Rôle

Figure 10.1 — Un échantillon de la base *Films*

10.1 SÉLECTIONS

Les requêtes les plus simples –et les plus courantes– sont celles qui recherchent, dans une table, des lignes satisfaisant un ou plusieurs critères de sélection. Par exemple, on recherche les titres des films du genre « Drame ».

```
mysql> SELECT titre
    -> FROM   Film
    -> WHERE  genre = 'Drame';
+--------------------+
| titre              |
+--------------------+
| Van Gogh           |
| Kagemusha          |
+--------------------+
2 rows in set (0.02 sec)
```

La structure de base d'une requête SQL comprend trois clauses `SELECT`, `FROM` et `WHERE`.

- `FROM` indique la (ou les) tables dans lesquelles on trouve les attributs utiles à la requête. Un attribut peut être« utile » de deux manières (non exclusives) : (1) on souhaite afficher son contenu *via* `SELECT`, (2) on souhaite qu'il ait une valeur particulière (une constante ou la valeur d'un autre attribut) *via* `WHERE`.
- `SELECT` indique la liste des attributs constituant le résultat.

- `WHERE` indique les conditions que doivent satisfaire les lignes de la table pour faire partie du résultat.

La clause `WHERE` est optionnelle : toutes les lignes de la tables sont sélectionnées si elle est omise. Voici donc la plus simple des requêtes : elle affiche toute la table.

```
mysql> SELECT * FROM Film;
+---------------------+-------+----------------+----------+
| titre               | annee | id_realisateur | genre    |
+---------------------+-------+----------------+----------+
| Impitoyable         |  1992 |             20 | Western  |
| Van Gogh            |  1990 |             29 | Drame    |
| Kagemusha           |  1980 |             68 | Drame    |
| Les pleins pouvoirs |  1997 |             20 | Policier |
+---------------------+-------+----------------+----------+
```

Un des problèmes rencontrés quand on commence à utiliser SQL (et même beaucoup plus tard ...) est de bien comprendre ce que signifie une requête, quelle que soit sa complexité. Quand il n'y a qu'une table – comme par exemple pour la requête sélectionnant les films dont le genre est « Drame » – l'interprétation est simple : on parcourt les lignes de la table *Film*. Pour chaque ligne, si l'attribut `genre` a pour valeur « Drame », on place l'attribut `titre` dans le résultat. Même si cette interprétation peut paraître élémentaire, elle devient très utile quand on a plusieurs tables dans le `FROM`. Une remarque en passant : il s'agit d'une manière *d'expliquer* la requête, ce qui ne signifie pas du tout que MySQL *l'exécute* de cette façon.

> **REMARQUE –** Rappelons que, sous Unix, MySQL distingue majuscules et minuscules dans le nom des tables.

10.1.1 Renommage, fonctions et constantes

Le résultat d'une requête SQL est *toujours* une table. On peut considérer en première approche que le calcul consiste à « découper », horizontalement et verticalement, la table indiquée dans le `FROM`. On peut aussi :

- renommer les attributs ;
- appliquer des fonctions aux attributs de chaque ligne ;
- introduire des constantes.

Fonctions MySQL

Les fonctions applicables aux valeurs des attributs sont par exemple les opérations arithmétiques (+, *, ...) pour les attributs numériques, les manipulations de chaînes de caractères (concaténation, sous-chaînes, mise en majuscules, ...). MySQL propose un ensemble très riche de fonctions (voir annexe B). Nous proposons quelques exemples ci-dessous.

- Donner la longueur des titres des films.

```
mysql> SELECT LENGTH(titre) FROM Film;
+---------------+
| LENGTH(titre) |
+---------------+
|            11 |
|             8 |
|             9 |
|            19 |
+---------------+
4 rows in set (0.00 sec)
```

Pour faciliter l'analyse syntaxique d'une requête, MySQL interdit tout blanc entre le nom de la fonction et la parenthèse ouvrante.

- Donner les 3 premières lettres du titre, concaténées avec l'année.

```
mysql> SELECT CONCAT(SUBSTRING(titre,1,3),annee) FROM Film;
+-----------------------------------+
| CONCAT(SUBSTRING(titre,1,3),annee) |
+-----------------------------------+
| Imp1992                           |
| Van1990                           |
| Kag1980                           |
| Les1997                           |
+-----------------------------------+
```

La norme SQL préconise « || » pour exprimer la concaténation, mais MySQL a choisi d'utiliser ce symbole pour le « ou » logique.

- Donner le nom des artistes et leur âge (arrondi grosièrement).

```
mysql> SELECT nom, YEAR(SYSDATE()) - annee_naissance FROM Artiste;
+----------+-----------------------------------+
| nom      | YEAR(SYSDATE()) - annee_naissance |
+----------+-----------------------------------+
| Eastwood |                                74 |
| Hackman  |                                74 |
| Pialat   |                                79 |
| Dutronc  |                              NULL |
| Kurosawa |                                94 |
+----------+-----------------------------------+
```

Les fonctions de manipulation de date constituent (avec celles consacrées aux chaînes de caractères) une large partie des fonctions MySQL. `SYSDATE()` donne la date courante, au format standard `AAAA-MM-JJ HH:MM:SS`, et `YEAR()` renvoie l'année.

- Finalement, on peut utiliser SQL pour exécuter des fonctions, sans sélectionner des lignes dans une table. Dans ce cas, le `FROM` est inutile (il s'agit d'une spécificité de MySQL). La requête suivante ajoute 3 mois à la date courante, à l'aide de la fonction `DATE_ADD()`.

```
mysql> select  DATE_ADD(NOW(), INTERVAL 3 MONTH);
+-----------------------------------+
| DATE_ADD(NOW(), INTERVAL 3 MONTH) |
+-----------------------------------+
| 2005-02-23 18:43:01               |
+-----------------------------------+
```

Renommage

Les noms des attributs sont par défaut ceux indiqués dans la clause `SELECT`, même en présence d'expressions complexes. L'expression `YEAR(SYSDATE()) - annee_naissance` peut donc tenir lieu de nom d'attribut pour le résultat, ce qui est peu pratique. Pour renommer les attributs, on utilise le mot-clé `AS`. Ce mot-clé est optionnel pour MySQL.

```
mysql> SELECT nom, YEAR(SYSDATE()) - annee_naissance AS age FROM Artiste;
+----------+------+
| nom      | age  |
+----------+------+
| Eastwood |   70 |
| Hackman  |   70 |
| Pialat   |   75 |
| Dutronc  | NULL |
| Kurosawa |   90 |
+----------+------+
```

On remarque que le calcul, appliqué à un `NULL`, donne un `NULL`. Nous reviendrons sur la gestion des `NULL` plus loin.

Constantes

On peut combiner, dans la clause `SELECT`, les noms des attributs de la table du `FROM` avec des constantes, ou *littéraux*, dont la valeur sera donc identique sur chaque ligne du résultat. Voici deux exemples, le second créant une ancre HTML à partir du contenu de la table.

```
mysql> SELECT 'Cette ligne correspond au film ', titre FROM Film;
+---------------------------------+---------------------+
| Cette ligne correspond au film  | titre               |
+---------------------------------+---------------------+
| Cette ligne correspond au film  | Impitoyable         |
| Cette ligne correspond au film  | Van Gogh            |
| Cette ligne correspond au film  | Kagemusha           |
| Cette ligne correspond au film  | Les pleins pouvoirs |
+---------------------------------+---------------------+
```

```
mysql> SELECT CONCAT('<a href="Acteur.php?nom=',
    ->            nom, '">Nom</a>') AS AncreActeur FROM Artiste;
+-----------------------------------------+
| AncreActeur                             |
+-----------------------------------------+
| <a href="Acteur.php?nom=Eastwood">Nom</a> |
| <a href="Acteur.php?nom=Hackman">Nom</a>  |
| <a href="Acteur.php?nom=Pialat">Nom</a>   |
| <a href="Acteur.php?nom=Dutronc">Nom</a>  |
| <a href="Acteur.php?nom=Kurosawa">Nom</a> |
+-----------------------------------------+
```

On peut introduire des apostrophes doubles (") dans une chaîne de caractères encadrée par des guillemets simples.

10.1.2 La clause DISTINCT

L'utilisation des clés permet d'éviter les doublons dans les tables stockées, mais il peuvent apparaître dans le résultat d'une requête. La clause `DISTINCT`, placée après le `SELECT`, permet de supprimer ces doublons. Voici deux exemples, avec et sans `DISTINCT`.

```
mysql>  SELECT annee_naissance FROM Artiste;
+-----------------+
| annee_naissance |
+-----------------+
|            1930 |
|            1930 |
|            1925 |
|            NULL |
|            1910 |
+-----------------+

mysql> SELECT DISTINCT annee_naissance FROM Artiste;
+-----------------+
| annee_naissance |
+-----------------+
|            NULL |
|            1910 |
|            1925 |
|            1930 |
+-----------------+
```

On trouve deux fois la valeur 1930 dans le premier résultat, et une seule fois dans le second. Il est également intéressant de constater que le second résultat est présenté en ordre croissant. De fait, la clause `DISTINCT` implique un tri préalable des lignes du résultat qui rassemble les doublons et permet de les éliminer facilement. Une conséquence, dont il faut savoir tenir compte, est que l'élimination des doublons peut être une opération coûteuse si le résultat est de taille importante.

10.1.3 La clause ORDER BY

Il est possible de trier le résultat d'une requête avec la clause ORDER BY suivie de la liste des attributs servant de critères au tri, même si ces derniers n'apparaissent pas dans le SELECT. Par défaut, le tri se fait en ordre croissant. On peut ajouter le mot-clé DESC après la liste des attributs pour demander un tri en ordre décroissant.

```
mysql> SELECT titre, genre FROM Film ORDER BY genre, annee;
+---------------------+----------+
| titre               | genre    |
+---------------------+----------+
| Kagemusha           | Drame    |
| Van Gogh            | Drame    |
| Les pleins pouvoirs | Policier |
| Impitoyable         | Western  |
+---------------------+----------+
```

Le tri sur le le résultat d'une fonction nécessite d'appliquer la fonction dans la clause SELECT, et de donner un nom à l'attribut obtenu avec AS.

```
mysql> SELECT nom, YEAR(SYSDATE()) - annee_naissance AS age
    -> FROM   Artiste
    -> ORDER BY age;
+----------+------+
| nom      | age  |
+----------+------+
| Dutronc  | NULL |
| Eastwood |   70 |
| Hackman  |   70 |
| Pialat   |   75 |
| Kurosawa |   90 |
+----------+------+
```

On peut trier aléatoirement le résultat d'une requête avec ORDER BY RAND(), RAND() étant la fonction qui engendre des nombres aléatoirement.

La clause ORDER BY, optionnelle, est toujours la dernière dans un ordre SQL.

10.1.4 La clause WHERE

Dans la clause WHERE, on spécifie une condition portant sur les attributs des tables du FROM. On utilise pour cela de manière standard le AND, le OR, le NOT et les parenthèses pour changer l'ordre de priorité des opérateurs booléens.

```
mysql> SELECT titre, annee
    -> FROM   Film
    -> WHERE  genre = 'Policier'
    -> AND    (annee > 1990 AND annee  <= 1999) ;
+---------------------+-------+
| titre               | annee |
+---------------------+-------+
| Les pleins pouvoirs |  1997 |
+---------------------+-------+
```

Les opérateurs de comparaison sont <, <=, >, >=, =, et <> pour exprimer la différence (!= est également possible). Pour obtenir une recherche par intervalle, on peut également utiliser le mot-clé `BETWEEN`. La requête précédente est équivalente à :

```
SELECT titre, annee
FROM   Film
WHERE  genre = 'Policier'
AND    annee BETWEEN 1990 AND 1999
```

Voici une requête avec `OR` et `NOT` : on recherche les films qui ne sont ni des drames, ni des policiers.

```
mysql> SELECT * FROM Film
    -> WHERE NOT (genre ='Drame' OR genre='Policier');
+-------------+-------+----------------+---------+
| titre       | annee | id_realisateur | genre   |
+-------------+-------+----------------+---------+
| Impitoyable |  1992 |             20 | Western |
+-------------+-------+----------------+---------+
```

Le `OR`, qui permet d'accepter plusieurs valeurs pour un attribut, peut s'exprimer plus simplement en rassemblant ces valeurs dans un ensemble, et en indiquant avec `IN` que cet attribut doit en faire partie.

```
mysql> SELECT * FROM Film
    -> WHERE genre NOT IN ('Drame','Policier');
+-------------+-------+----------------+---------+
| titre       | annee | id_realisateur | genre   |
+-------------+-------+----------------+---------+
| Impitoyable |  1992 |             20 | Western |
+-------------+-------+----------------+---------+
```

On peut effectuer des comparaisons non seulement entre un attribut et une constante, comme dans les exemples ci-dessus, mais également entre deux attributs. Voici la requête qui sélectionne tous les films dans lesquels un rôle et le titre du film sont identiques.

```
mysql> SELECT *
    -> FROM Role
    -> Where titre=nom_role;
+----------+-----------+----------+
| titre    | id_acteur | nom_role |
+----------+-----------+----------+
| Van Gogh |        30 | Van Gogh |
+----------+-----------+----------+
```

Chaînes de caractères

Les comparaisons de chaînes de caractères soulèvent quelques problèmes délicats.

1. Attention aux différences entre chaînes de longueur fixe et chaînes de longueur variable. Les premières sont complétées par des blancs (' '), pas les secondes.
2. Si les chaînes de caractères sont de type `BINARY VARCHAR` ou `BLOB`, MySQL distingue majuscules et minuscules, et 'IMPITOYABLE' est considéré comme différent de 'Impitoyable'. C'est d'ailleurs le comportement par défaut d'autres SGBD, comme ORACLE.

Il est préférable d'utiliser toujours des chaînes de longueur variable, de type `VARCHAR` ou `TEXT`. Les opérateurs de comparaison donnent alors les résultats attendus intuitivement.

Les recherches sur chaînes de caractères demandent des fonctionnalités plus étendues que celles sur les numériques. MySQL fournit des options pour les recherches par motif (*pattern matching*) à l'aide de la clause `LIKE`, conformes à la norme SQL ANSI :

1. le caractère « _ » désigne n'importe quel caractère ;
2. le caractère « %' » désigne n'importe quelle chaîne de caractères.

Par exemple, voici la requête cherchant tous les artistes dont la deuxième lettre du nom est un 'a'.

```
mysql> SELECT * FROM Artiste WHERE nom LIKE '_a%';
+----+----------+--------+-----------------+
| id | nom      | prenom | annee_naissance |
+----+----------+--------+-----------------+
| 20 | Eastwood | Clint  |            1930 |
| 21 | Hackman  | Gene   |            1930 |
+----+----------+--------+-----------------+
```

Quels sont les titres de film qui ne contiennent pas le caractère blanc ?

```
mysql>  SELECT * FROM Film WHERE titre NOT LIKE '% %';
+-------------+-------+----------------+---------+
| titre       | annee | id_realisateur | genre   |
+-------------+-------+----------------+---------+
| Impitoyable |  1992 |             20 | Western |
| Kagemusha   |  1980 |             68 | Drame   |
+-------------+-------+----------------+---------+
```

10.1.5 Dates

Tous les SGBD proposaient bien avant la normalisation leur propre format de date, et la norme n'est de ce fait pas suivie par tous. MySQL est assez proche de la norme SQL ANSI, en raison de son apparition relativement tardive.

Une date est spécifiée par une chaîne de caractères au format 'AAAA-MM-JJ', par exemple `'2004-03-01'` pour le premier mars 2004. Les zéros sont nécessaires afin que le mois et le quantième du jour comprennent systématiquement deux chiffres, mais le tiret est optionnel. On peut compléter une date avec l'horaire au format 'HH:MM:SS', ce qui correspond à une valeur du type `DATETIME`. MySQL fait son possible pour interpréter correctement la chaîne de caractères proposée, et convertit une `DATE` en `DATETIME` ou inversement, selon les besoins.

On peut effectuer des sélections sur les dates à l'aide des comparateurs usuels. Sur la table *Message* de la base *Film*, voici comment sélectionner les messages du 3 avril 2004.

```
SELECT * FROM Message WHERE date_creation = '2004-04-03'
```

L'attribut `date_creation`, de type `DATETIME`, est converti en `DATE` (par suppression de l'horaire) pour être comparé à '2004-04-03'.

MySQL propose de nombreuses fonctions permettant de calculer des écarts de dates, d'ajouter des mois ou des années à des dates, etc : voir l'annexe B.

10.1.6 Valeurs nulles

La valeur de certains attributs peut être inconnue ; on parle alors de *valeur nulle*, désignée par le mot-clé `NULL`. Il est très important de comprendre que la « valeur nulle » n'est pas une valeur mais une *absence* de valeur, et que l'on ne peut lui appliquer aucune des opérations ou comparaisons usuelles. En conséquence,

- toute opération ou fonction appliquée à `NULL` donne pour résultat `NULL` ;
- toute comparaison avec `NULL` donne un résultat qui n'est ni vrai, ni faux mais une troisième valeur booléenne, `UNKNOWN`.

La présence de NULL peut produire des effets surprenants. Par exemple, la requête suivante

```
SELECT *
FROM   Artiste
WHERE  annee_naissance <= 1950 OR annee_naissance >= 1950
```

devrait en principe ramener tous les artistes. En fait, 'Dutronc' ne figurera pas dans le résultat car annee_naissance est à NULL, et la comparaison a pour résultat UNKNOWN.

Autre piège : NULL est un mot-clé, pas une constante. Une comparaison comme annee_naissance = NULL ne sera pas correctement interprétée par MySQL.

```
mysql> select * from Artiste WHERE annee_naissance = NULL;
Empty set (0.00 sec)
```

Le test correct de l'absence de valeur dans une colonne est *x* IS NULL, l'inverse étant IS NOT NULL).

```
mysql> select * from Artiste WHERE annee_naissance IS NULL;
+----+---------+---------+-----------------+
| id | nom     | prenom  | annee_naissance |
+----+---------+---------+-----------------+
| 30 | Dutronc | Jacques |            NULL |
+----+---------+---------+-----------------+
```

Dès que l'on commence à exprimer des clauses WHERE compliquées, la présence de NULL dans la table devient difficile à manipuler. Il existe, dans la norme SQL ANSI, des règles très précises pour la prise en compte des NULL, basées les valeurs suivantes pour TRUE, FALSE et UNKNOWN : TRUE vaut 1, FALSE 0 et UNKNOWN 1/2. Les connecteurs logiques donnent alors les résultats suivants :

1. x AND $y = min(x, y)$
2. x OR $y = max(x, y)$
3. NOT $x = 1 - x$

Les conditions exprimées dans une clause WHERE sont évaluées pour chaque ligne, et ne sont conservées dans le résultat que les lignes pour lesquelles cette évaluation donne TRUE. Reprenons la requête déjà vue ci-dessus.

```
SELECT *
FROM   Artiste
WHERE  annee_naissance <= 1950 OR annee_naissance >= 1950
```

La ligne « Jacques Dutronc » n'a pas de valeur pour annee_naissance. Les deux comparaisons du WHERE ont pour valeur 1/2, et le OR a pour valeur $max(1/2, 1/2) = 1/2$, ce qui explique que la ligne ne fasse pas partie du résultat.

En résumé, le NULL est une source de problèmes : dans la mesure du possible il faut l'éviter au moment où on définit la table en spécifiant la contrainte NOT NULL ou en imposant une valeur par défaut. Si la valeur par défaut n'est pas fixée à la création de la table, on peut en donner une au moment de l'exécution de la requête avec la fonction IFNULL() qui remplace une valeur NULL par son deuxième argument.

```
mysql> SELECT IFNULL(annee_naissance, 'Pas de date de naissance !?')
    -> FROM Artiste;
+-----------------------------------------------------+
| IFNULL(annee_naissance, 'Pas de date de naissance !?') |
+-----------------------------------------------------+
| 1930                                                |
| 1930                                                |
| 1925                                                |
| Pas de date de naissance !?                         |
| 1910                                                |
+-----------------------------------------------------+
```

10.1.7 Clauses spécifiques à MySQL

La clause LIMIT peut être placée à la fin de toute requête SQL, et indique le nombre maximal de lignes dans le résultat.

```
mysql> SELECT *
    -> FROM Film
    -> LIMIT 3;
+-------------+-------+----------------+---------+
| titre       | annee | id_realisateur | genre   |
+-------------+-------+----------------+---------+
| Impitoyable |  1992 |             20 | Western |
| Van Gogh    |  1990 |             29 | Drame   |
| Kagemusha   |  1980 |             68 | Drame   |
+-------------+-------+----------------+---------+
```

Si on utilise deux chiffres comme, par exemple, LIMIT 1,3, le premier indique le numéro de la ligne à partir de laquelle la limite s'applique, les lignes étant numérotées à partir de 0.

```
mysql> SELECT *
    -> FROM Film
    -> LIMIT 1,3;
+--------------------+-------+----------------+----------+
| titre              | annee | id_realisateur | genre    |
+--------------------+-------+----------------+----------+
| Van Gogh           |  1990 |             29 | Drame    |
| Kagemusha          |  1980 |             68 | Drame    |
| Les pleins pouvoirs |  1997 |             20 | Policier |
+--------------------+-------+----------------+----------+
```

Au lieu d'afficher à l'écran, on peut placer le résultat d'un ordre `SELECT` dans un fichier :

```
SELECT *
INTO OUTFILE './svfilm.txt'
FROM Film
```

Il faut disposer du privilège `file` pour utiliser `INTO OUTFILE`. On obtient alors tous les droits d'accès du serveur *mysqld*.

Par défaut, le fichier est créé dans le répertoire contenant les bases de données. On peut indiquer explicitement le chemin d'accès, à condition que le programme client *mysql* ait le droit d'écriture.

Les lignes de la table sont écrites en séparant chaque valeur par une tabulation, ce qui permet de recharger le fichier par la suite avec la commande `LOAD DATA` (voir la section présentant cette commande, page 29). On peut indiquer, après le nom du fichier, les options de séparation des lignes et des attributs, avec une syntaxe identique à celle utilisée pour `LOAD DATA` : voir annexe B.

10.2 JOINTURES

La jointure est une des opérations les plus utiles (et l'une des plus courantes) puisqu'elle permet d'exprimer des requêtes portant sur des données réparties dans plusieurs tables. La syntaxe pour exprimer des jointures avec SQL est une extension directe de celle étudiée précédemment dans le cas des sélections simples : on donne la liste des tables concernées dans la clause `FROM`, et on exprime les critères de rapprochement entre ces tables dans la clause `WHERE`.

10.2.1 Interprétation d'une jointure

Prenons l'exemple de la requête donnant le titre des films avec le nom et le prénom de leur metteur en scène.

```
mysql> SELECT titre, prenom, nom
    -> FROM   Film, Artiste
    -> WHERE  id_realisateur = id;
+---------------------+---------+----------+
| titre               | prenom  | nom      |
+---------------------+---------+----------+
| Impitoyable         | Clint   | Eastwood |
| Les pleins pouvoirs | Clint   | Eastwood |
| Van Gogh            | Maurice | Pialat   |
| Kagemusha           | Akira   | Kurosawa |
+---------------------+---------+----------+
```

Pour bien comprendre ce que signifie une jointure, ce qui est parfois difficile quand on commence à utiliser SQL, on peut généraliser l'interprétation donnée

dans le cas d'une seule table. La clause `FROM`, dans les deux cas, définit un « espace de recherche » qui, quand il y a une seule table, correspond à toutes les lignes de celle-ci. Quand il y a deux tables, cet espace de recherche est constitué de toutes les combinaisons possibles des lignes des deux tables.

La figure 10.2 montre toutes ces combinaisons sous la forme d'une table, comprenant $4 \times 5 = 20$ lignes construites à partir des 4 lignes de *Film* et des cinq lignes de *Artiste*. Appelons-la *FilmXArtiste* : elle peut être obtenue avec la requête.

```
SELECT titre, prenom, nom
FROM   Film, Artiste
```

Dans cette table, beaucoup de lignes ne nous intéressent pas, comme celles qui associent Pialat et *Impitoyable*, ou *Van Gogh* et Kurosawa. En fait on ne veut garder que celles pour lesquelles l'attribut `id_realisateur` est égal à l'attribut `id`, soit 4 lignes.

titre	annee	id_real.	genre	id	nom	prenom	annee_naiss.
Impitoyable	1992	20	Western	20	Eastwood	Clint	1930
Impitoyable	1992	20	Western	21	Hackman	Gene	1930
Impitoyable	1992	20	Western	29	Pialat	Maurice	1925
Impitoyable	1992	20	Western	30	Dutronc	Jacques	
Impitoyable	1992	20	Western	68	Kurosawa	Akira	1910
Van Gogh	1990	29	Drame	20	Eastwood	Clint	1930
Van Gogh	1990	29	Drame	21	Hackman	Gene	1930
Van Gogh	1990	29	Drame	29	Pialat	Maurice	1925
Van Gogh	1990	29	Drame	30	Dutronc	Jacques	
Van Gogh	1990	29	Drame	68	Kurosawa	Akira	1910
Kagemusha	1980	68	Drame	20	Eastwood	Clint	1930
Kagemusha	1980	68	Drame	21	Hackman	Gene	1930
Kagemusha	1980	68	Drame	29	Pialat	Maurice	1925
Kagemusha	1980	68	Drame	30	Dutronc	Jacques	
Kagemusha	1980	68	Drame	68	Kurosawa	Akira	1910
Les pleins pouvoirs	1997	20	Policier	20	Eastwood	Clint	1930
Les pleins pouvoirs	1997	20	Policier	21	Hackman	Gene	1930
Les pleins pouvoirs	1997	20	Policier	29	Pialat	Maurice	1925
Les pleins pouvoirs	1997	20	Policier	30	Dutronc	Jacques	
Les pleins pouvoirs	1997	20	Policier	68	Kurosawa	Akira	1910

Figure 10.2 — Table *FilmXArtiste,* définie par la clause `FROM Film, Artiste`.

La jointure est simplement une sélection sur cette table *FilmXArtiste*, que l'on pourrait exprimer de la manière suivante si cette table existait.

```
SELECT titre, prenom, nom
FROM   FilmXArtiste
WHERE  id_realisateur = id
```

L'interprétation d'une jointure est donc une généralisation de l'interprétation d'un ordre SQL portant sur une seule table. On parcourt toutes les lignes définies par la clause `FROM`, et on leur applique la condition exprimée dans le `WHERE`. Finalement, on ne garde que les attributs spécifiés dans la clause `SELECT`. C'est vrai quel que soit le nombre de tables utilisées dans le `FROM`.

Une remarque importante pour finir : la jointure est une opération qui consiste à reconstituer une association entre entités (voir chapitre 4), dans notre exemple l'association entre un film et son metteur en scène. Comme nous avons vu que cette association était représentée dans un schéma relationnel par le mécanisme de clés primaires et clés étrangères, la plupart des jointures s'expriment par une égalité entre la clé primaire d'une table et la clé étrangère correspondante dans l'autre table. C'est le cas dans l'exemple ci-dessus, où `id_realisateur` est la clé étrangère, dans *Film*, correspondant à la clé primaire `id` dans *Artiste*.

10.2.2 Gestion des ambiguïtés

Dans l'exemple précédent, il n'y a pas d'ambiguïté sur les noms des attributs : `titre` et `id_realisateur` viennent de la table `Film`, tandis que `nom, prenom` et `id` viennent de la table `Artiste`. Il peut arriver (il arrive de fait fréquemment) qu'un même nom d'attribut soit partagé par plusieurs tables impliquées dans une jointure. Dans ce cas, on résout l'ambiguïté en préfixant l'attribut par le nom de sa table.

Exemple : afficher, pour chaque film, les rôles du film.

```
mysql> SELECT Film.titre, nom_role
    -> FROM   Film, Role
    -> WHERE  Film.titre = Role.titre;
+---------------------+--------------------+
| titre               | nom_role           |
+---------------------+--------------------+
| Impitoyable         | William Munny      |
| Impitoyable         | Little Bill Dagget |
| Van Gogh            | Van Gogh           |
| Les pleins pouvoirs | Le président       |
+---------------------+--------------------+
```

Il n'y a pas ici de problème pour `nom_role` qui désigne sans ambiguïté possible un attribut de la table *Role*. Si, en revanche, on ne préfixe pas `titre` par la table dont il provient, MySQL ne sait pas évaluer la requête.

```
mysql> SELECT titre, nom_role
    -> FROM   Film, Role
    -> WHERE  titre = titre;
ERROR 1052: Column: 'titre' in field list is ambiguous
```

Comme il peut être fastidieux de répéter intégralement le nom d'une table, on peut lui associer un synonyme et utiliser ce synonyme en tant que préfixe. La requête précédente devient par exemple :

```
SELECT f.titre, nom_role
FROM   Film AS f, Role AS r
WHERE  f.titre = r.titre
```

Pour obtenir le nom de l'acteur qui a joué le rôle, il faut effectuer une jointure supplémentaire avec la table *Artiste*.

```
mysql> SELECT f.titre, prenom, nom, nom_role
    -> FROM   Film AS f, Role AS r, Artiste
    -> WHERE  f.titre = r.titre
    -> AND    id_acteur = id;
+--------------------+---------+----------+--------------------+
| titre              | prenom  | nom      | nom_role           |
+--------------------+---------+----------+--------------------+
| Impitoyable        | Clint   | Eastwood | William Munny      |
| Impitoyable        | Gene    | Hackman  | Little Bill Dagget |
| Van Gogh           | Jacques | Dutronc  | Van Gogh           |
| Les pleins pouvoirs | Gene    | Hackman  | Le président       |
+--------------------+---------+----------+--------------------+
```

On a une jointure entre *Film* et *Role*, une autre entre *Role* et *Artiste*. En ajoutant une jointure entre *Artiste* et *Film*, on obtient les metteurs en scène qui ont joué dans leur propre film.

```
mysql> SELECT f.titre, prenom, nom, nom_role
    -> FROM   Film AS f, Role AS r, Artiste
    -> WHERE  f.titre = r.titre
    -> AND    id_acteur = id
    -> AND     id = id_realisateur;
+-------------+--------+----------+---------------+
| titre       | prenom | nom      | nom_role      |
+-------------+--------+----------+---------------+
| Impitoyable | Clint  | Eastwood | William Munny |
+-------------+--------+----------+---------------+
```

Il n'y a pas d'ambiguïté sur les noms d'attributs, à part pour `titre`, donc il est inutile en l'occurrence d'employer des synonymes. Il existe en revanche une situation où l'utilisation des synonymes est indispensable : celle ou l'on souhaite effectuer une jointure d'une table avec elle-même.

Considérons la requête suivante : *Donner les paires d'artistes qui sont nés la même année*. Ici toutes les informations nécessaires sont dans la seule table `Artiste`, mais on construit une ligne dans le résultat avec *deux lignes* de la table. Tout se passe comme si on devait faire la jointure entre deux versions distinctes de la table `Artiste`. On résout le problème en utilisant deux synonymes distincts (nous omettons le mot-clé `AS` qui est optionnel).

```
mysql> SELECT a1.nom, a2.nom
    -> FROM   Artiste a1, Artiste a2
    -> WHERE  a1.annee_naissance = a2.annee_naissance;
+----------+----------+
| nom      | nom      |
+----------+----------+
| Eastwood | Eastwood |
| Hackman  | Eastwood |
| Eastwood | Hackman  |
| Hackman  | Hackman  |
| Pialat   | Pialat   |
| Kurosawa | Kurosawa |
+----------+----------+
```

Le résultat obtenu est techniquement correct, mais cela ne nous intéresse pas de savoir qu'un artiste est né la même année que lui-même, ou d'avoir les paires [Hackman, Eastwood], puis [Eastwood, Hackman]. Pour éliminer les lignes inutiles, il suffit d'enrichir un peu la clause `WHERE`.

```
mysql> SELECT a1.nom, a2.nom
    -> FROM   Artiste a1, Artiste a2
    -> WHERE  a1.annee_naissance = a2.annee_naissance
    -> AND    a1.id < a2.id;
+----------+---------+
| nom      | nom     |
+----------+---------+
| Eastwood | Hackman |
+----------+---------+
```

On peut imaginer que `a1` et `a2` sont deux « curseurs » qui parcourent indépendamment la table `Artiste` et permettent de constituer des couples de lignes auxquelles on applique la condition de jointure.

Si on recherche maintenant les films avec leur metteur en scène, ainsi que les acteurs qui y ont joué un rôle, on obtient la requête suivante.

```
mysql> SELECT f.titre, MES.nom AS nom_realisateur,
    ->              Acteur.nom AS nom_acteur, nom_role
    -> FROM   Film AS f, Role AS r, Artiste MES, Artiste Acteur
    -> WHERE  f.titre = r.titre
    -> AND    id_acteur = Acteur.id
    -> AND    MES.id = id_realisateur;
+--------------------+----------------+------------+--------------------+
| titre              | nom_realisateur | nom_acteur | nom_role           |
+--------------------+----------------+------------+--------------------+
| Impitoyable        | Eastwood       | Eastwood   | William Munny      |
| Impitoyable        | Eastwood       | Hackman    | Little Bill Dagget |
| Van Gogh           | Pialat         | Dutronc    | Van Gogh           |
| Les pleins pouvoirs | Eastwood       | Hackman    | Le président       |
+--------------------+----------------+------------+--------------------+
```

Le module de recherche du site *Films* (voir page 289) est basé sur des jointures et des sélections assez complexes, combinant les fonctionnalités vues jusqu'à présent.

10.2.3 Jointures externes

Quand on effectue la jointure entre *Film* et *Role* pour obtenir les rôles d'un film, rien n'apparaît pour le film *Kagemusha*.

```
mysql> SELECT f.titre, nom_role
    -> FROM   Film AS f, Role AS r
    -> WHERE  f.titre = r.titre;
+--------------------+--------------------+
| titre              | nom_role           |
+--------------------+--------------------+
| Impitoyable        | William Munny      |
| Impitoyable        | Little Bill Dagget |
| Van Gogh           | Van Gogh           |
| Les pleins pouvoirs | Le président      |
+--------------------+--------------------+
```

En effet, pour ce film, aucun rôle n'a été inséré dans la base de données. Pour éviter cet effet parfois indésirable, on peut effectuer une *jointure externe*. Ce type de jointure prend une table comme table directrice, conventionnellement considérée comme la table de gauche, et utilise la table de droite comme table optionnelle. Si, pour une ligne de la table de gauche, on trouve une ligne satisfaisant le critère de jointure dans la table de droite, alors la jointure s'effectue normalement. Sinon les attributs provenant de la table de droite sont affichés à `NULL`.

La clause de jointure externe est `LEFT OUTER JOIN` et le critère de jointure doit être précédé du mot-clé `ON`. Voici la jointure externe entre *Film* et *Role*. Le mot-clé `OUTER` est en optionnel.

```
mysql> SELECT Film.titre, nom_role
    -> FROM Film LEFT JOIN Role ON Film.titre=Role.titre;
+--------------------+--------------------+
| titre              | nom_role           |
+--------------------+--------------------+
| Impitoyable        | William Munny      |
| Impitoyable        | Little Bill Dagget |
| Van Gogh           | Van Gogh           |
| Kagemusha          | NULL               |
| Les pleins pouvoirs | Le président      |
+--------------------+--------------------+
```

Comme il est fréquent que la jointure porte sur des attributs portant le même nom dans les deux tables, MySQL (qui suit en cela la norme SQL ANSI bien mieux que la plupart des autres SGBD) propose une jointure dit « naturelle » qui s'effectue, implicitement, en testant l'égalité de tous les attributs communs aux deux tables. Dans l'exemple ci-dessous, la jointure se fait donc sur `titre`.

```
mysql> SELECT Film.titre, nom_role
    -> FROM Film NATURAL LEFT JOIN Role;
+---------------------+---------------------+
| titre               | nom_role            |
+---------------------+---------------------+
| Impitoyable         | William Munny       |
| Impitoyable         | Little Bill Dagget  |
| Van Gogh            | Van Gogh            |
| Kagemusha           | NULL                |
| Les pleins pouvoirs | Le président        |
+---------------------+---------------------+
```

On peut combiner la jointure externe avec des jointures normales, des sélections, des tris, etc.

```
mysql> SELECT f.titre, prenom, nom, nom_role
    -> FROM Film AS f LEFT OUTER JOIN Role AS r ON f.titre=r.titre,
    ->          Artiste
    -> WHERE  annee < 1995
    -> AND    id = id_acteur
    -> ORDER BY annee;
+-------------+---------+----------+---------------------+
| titre       | prenom  | nom      | nom_role            |
+-------------+---------+----------+---------------------+
| Van Gogh    | Jacques | Dutronc  | Van Gogh            |
| Impitoyable | Clint   | Eastwood | William Munny       |
| Impitoyable | Gene    | Hackman  | Little Bill Dagget  |
+-------------+---------+----------+---------------------+
```

Encore une fois le principe est toujours le suivant : la clause `FROM` définit l'espace de recherche (un ensemble de lignes obtenues par combinaison des tables apparaissant dans le `FROM`), le `WHERE` sélectionne des lignes, et le `SELECT` des colonnes.

10.3 OPÉRATIONS ENSEMBLISTES

La norme SQL ANSI comprend des opérations qui considèrent les tables comme des ensembles, et effectuent des intersections, des unions ou des différences avec les mot-clé `UNION`, `INTERSECT` ou `EXCEPT`. Chaque opérateur s'applique à deux tables de schéma identique (même nombre d'attributs, mêmes noms, mêmes types). Trois exemples suffiront pour illustrer ces opérations.

1. Donnez toutes les années dans la base.

```
SELECT annee FROM Film
 UNION
SELECT annee_naissance AS annee FROM Artiste
```

2. Donnez les noms de rôles qui sont aussi des titres de films.

```
SELECT nom_role AS nom FROM Role
 INTERSECT
SELECT titre  AS nom FROM Film
```

3. Quels sont les noms de rôles qui ne sont pas des titres de films ?

```
SELECT nom_role AS nom FROM Role
 EXCEPT
SELECT titre  AS nom FROM Film
```

L'opérateur `INTERSECT` s'exprime facilement avec une jointure. Le `EXCEPT` est important en principe car il permet d'exprimer des *négations*, à savoir toutes les requêtes où on effectue une recherche en prenant des lignes qui n'ont pas telle ou telle propriété : *les acteurs qui n'ont pas de rôle* ou *les films pour lesquels on ne connaît pas les acteurs*. On peut également exprimer des négations avec les clauses `NOT IN` et `NOT EXISTS`.

10.4 REQUÊTES IMBRIQUÉES

Toutes les requêtes qui suivent ne peuvent s'exprimer qu'à partir de la version 4.1 de MySQL.

Qu'est-ce qu'une requête imbriquée ? Jusqu'à présent, les conditions exprimées dans la clause `WHERE` consistaient en comparaisons d'attributs avec des valeurs scalaires, avec une exception : le mot-clé `IN` permet de tester l'appartenance de la valeur d'un attribut à un ensemble. La requête suivante donne tous les rôles des films de Clint Eastwood qui sont dans la base.

```
mysql> SELECT * FROM Role
    -> WHERE  titre IN ('Impitoyable','Les pleins pouvoirs');
+--------------------+-----------+--------------------+
| titre              | id_acteur | nom_role           |
+--------------------+-----------+--------------------+
| Impitoyable        |        20 | William Munny      |
| Impitoyable        |        21 | Little Bill Dagget |
| Les pleins pouvoirs |        21 | Le président       |
+--------------------+-----------+--------------------+
```

Les requêtes imbriquées sont une généralisation de cette construction : au lieu d'utiliser un ensemble de valeurs donné « en dur », on le construit dynamiquement avec une sous-requête. Dans une situation plus réaliste, on ne connaîtrait pas *a priori* les titres de tous les films de Clint Eastwood. On construit donc la liste des films avec une sous-requête.

```
SELECT * FROM Role
WHERE  titre IN (SELECT titre FROM Film, Artiste
                 WHERE  Film.id_realisateur=Artiste.id
                 AND    nom='Eastwood')
```

Le mot-clé `IN` exprime la *condition d'appartenance* de `titre` à la table formée par la requête imbriquée. Le principe général des requêtes imbriquées est d'exprimer des conditions sur des tables calculées par des requêtes. Ces conditions sont les suivantes :

1. `EXISTS R` : renvoie `TRUE` si *R* n'est pas vide, `FALSE` sinon.
2. *t* `IN R` où *t* est une ligne dont le type (le nombre et le type des attributs) est celui de *R* : renvoie `TRUE` si *t* appartient à *R*, `FALSE` sinon.
3. *v cmp* `ANY R`, où *cmp* est un comparateur SQL (<, >, =, etc.) : renvoie `TRUE` si la comparaison avec *au moins une* des lignes de la table *R* renvoie `TRUE`.
4. *v cmp* `ALL R`, où *cmp* est un comparateur SQL (<, >, =, etc.) : renvoie `TRUE` si la comparaison avec *toutes* les lignes de la table *R* renvoie `TRUE`.

Toutes ces expressions peuvent être préfixées par `NOT` pour obtenir la négation. La richesse des expressions possibles permet d'effectuer une même interrogation en choisissant parmi plusieurs syntaxes. En général, tout ce qui n'est pas basé sur une négation `NOT IN` ou `NOT EXISTS` peut s'exprimer *sans* requête imbriquée. Voici maintenant quelques exemples.

10.4.1 Exemples de requêtes imbriquées

Reprenons la requête donnant les rôles des films de Clint Eastwood. On peut l'exprimer avec une requête imbriquée, et la comparaison `= ANY` qui est synonyme de `IN`.

```
SELECT * FROM Role
WHERE  titre = ANY (SELECT titre FROM Film, Artiste
                    WHERE  Film.id_realisateur=Artiste.id
                    AND    nom='Eastwood')
```

Il est très important de noter que cette requête (et donc celle avec `IN`) est *équivalente* [1] à la jointure suivante :

```
mysql> SELECT r.*
    -> FROM Role r, Film f, Artiste a
    -> WHERE r.titre = f.titre
    -> AND  a.id= f.id_realisateur
    -> AND  nom='Eastwood';
+---------------------+----------+---------------------+
| titre               | id_acteur | nom_role            |
+---------------------+----------+------__--------------+
| Impitoyable         |       20 | William Munny       |
| Impitoyable         |       21 | Little Bill Dagget  |
| Les pleins pouvoirs |       21 | Le président        |
+---------------------+----------+---------------------+
```

1. Deux requêtes sont équivalentes si elles donnent toujours le même résultat, quelle que soit la base.

Notez la syntaxe `table.*` qui représente tous les attributs de la table *table*. On peut considérer que la jointure est moins naturelle, et que la requête imbriquée est plus proche de la manière dont la recherche est conçue : on ne s'intéresse pas directement aux films de Clint Eastwood, mais seulement aux rôles. Il n'en reste pas moins que toutes deux donnent le même résultat. Autre exemple : *donner les films pour lesquels on connaît au moins un des rôles*. On peut utiliser une requête imbriquée.

```
SELECT * FROM Film
WHERE  titre IN (SELECT titre FROM Role)
```

On va donc parcourir les films, et pour chacun, on affichera son titre si et seulement si ce titre apparaît dans au moins une des lignes de la table *Role*. On peut là aussi utiliser une jointure.

```
SELECT  DISTINCT Film.*
FROM Film, Role
WHERE  Film.titre = Role.titre
```

Il y a une différence un peu subtile : avec la jointure on affichera autant de fois un titre qu'il y a de rôles. Le mot-clé `DISTINCT` permet de se ramener à un résultat équivalent à celui de la requête imbriquée.

On peut exprimer la condition d'appartenance sur des lignes comprenant plusieurs attributs, comme le montre la requête suivante : on recherche tous les films du même genre qu'*Impitoyable*, et sont parus la même année.

```
SELECT *
FROM Film
WHERE (annee, genre) = (SELECT annee, genre
                        FROM Film f
                        WHERE titre='Impitoyable')
```

Le nombre et le type des attributs (ici deux attributs) doit correspondre exactement dans la requête principale et la requête imbriquée.

Bien entendu la requête ci-dessus s'exprime avec une jointure. Ce n'est pas le cas en revanche de celle ci-dessous, qui sélectionne l'artiste avec la date de naissance la plus ancienne.

```
SELECT prenom, nom
FROM Artiste
WHERE annee_naissance <= ALL (SELECT annee_naissance FROM Artiste
                        WHERE annee_naissance IS NOT NULL)
AND   annee_naissance IS NOT NULL;

+---------+----------+
| prenom  | nom      |
+---------+----------+
| Akira   | Kurosawa |
+---------+----------+
```

Le ALL exprime une comparaison qui vaut pour *toutes* les lignes ramenées par la requête imbriquée. Attention aux valeurs à NULL dans ce genre de situation : toute comparaison avec une de ces valeurs renvoie UNKNOWN et cela peut entraîner l'échec du ALL. Il n'existe pas d'expression avec jointure qui puisse exprimer ce genre de condition. En revanche, le ALL peut s'exprimer avec la négation, selon la règle d'équivalence que quand quelque chose est toujours vrai, il n'est jamais faux ! Nous verrons un exemple plus loin.

10.4.2 Requêtes corrélées

Les exemples de requêtes imbriquées donnés précédemment pouvaient être évalués indépendamment de la requête principale, ce qui permet au système (s'il le juge nécessaire) d'exécuter la requête en deux phases. La clause EXISTS fournit encore un nouveau moyen d'exprimer les requêtes vues précédemment, en basant la sous-requête sur une ou plusieurs valeurs issues de la requête principale. On parle alors de *requêtes corrélées*.

Reprenons une dernière fois la requête donnant les rôles des films de Clint Eastwood. Elle s'exprime avec EXISTS de la manière suivante :

```
SELECT * FROM Role
WHERE  EXISTS  (SELECT titre FROM Film, Artiste
                WHERE  Film.id_realisateur=Artiste.id
                AND    Film.titre=Role.titre
                AND    nom='Eastwood')
```

```
+---------------------+----------+----------------------+
| titre               | id_acteur| nom_role             |
+---------------------+----------+----------------------+
| Impitoyable         |       20 | William Munny        |
| Impitoyable         |       21 | Little Bill Dagget   |
| Les pleins pouvoirs |       21 | Le président         |
+---------------------+----------+----------------------+
```

On obtient donc une nouvelle technique d'expression, qui permet d'aborder le critère de recherche sous une troisième perspective : on conserve un rôle si, *pour ce rôle*, le film a été dirigé par Clint Eastwood. Notez la jointure entre la table *Role* référencée dans la requête principale et la table *Film* de la requête imbriquée. C'est cette comparaison « à distance » entre deux tables référencées par des clauses FROM différentes qui explique le terme de *corrélation*.

Supposons que l'on veuille trouver tous les metteurs en scène ayant dirigé Gene Hackman. La requête peut s'exprimer avec EXIST de la manière suivante :

```
SELECT * FROM Artiste a1
WHERE  EXISTS  (SELECT *
                FROM Film f, Role r, Artiste a2
                WHERE f.titre = r.titre
                AND   r.id_acteur = a2.id
```

```
                AND   nom = 'Hackman'
                AND   f.id_realisateur = a1. id)

+----+----------+--------+-----------------+
| id | nom      | prenom | annee_naissance |
+----+----------+--------+-----------------+
| 20 | Eastwood | Clint  |       1930      |
+----+----------+--------+-----------------+
```

En langage naturel, le raisonnement est le suivant : on prend tous les artistes (requête principale) tels que, parmi les films qu'ils ont dirigés (requête secondaire), on trouve un rôle joué par Gene Hackman.

> **REMARQUE –** dans une sous-requête associée à la clause `EXISTS`, peu importent les attributs du `SELECT` puisque la condition se résume à : cette requête ramène-t-elle au moins une ligne ou non ? On peut donc systématiquement utiliser `SELECT *`.

La requête équivalente avec `IN` s'appuie sur un raisonnement légèrement modifié : on prend tous les artistes dont l'identifiant fait partie de l'ensemble des identifiants des metteurs en scène d'un film avec Gene Hackman.

```
SELECT * FROM Artiste a1
WHERE  id IN (SELECT id_realisateur
                FROM Film f, Role r, Artiste a2
                WHERE f.titre = r.titre
                AND   r.id_acteur = a2.id
                AND   nom = 'Hackman')
```

La solution classique d'une jointure « à plat » reste valable, en utilisant `DISTINCT` pour éliminer les doublons :

```
SELECT DISTINCT a1.*
FROM Artiste a1, Film f, Role r, Artiste a2
WHERE f.titre = r.titre
AND   r.id_acteur = a2.id
AND   a2.nom = 'Hackman'
AND   f.id_realisateur = a1. id
```

Enfin, rien n'empêche d'utiliser plusieurs niveaux d'imbrication !

```
SELECT * FROM Artiste a1
WHERE  EXISTS
      (SELECT * FROM Film f
           WHERE f.id_realisateur = a1. id
           AND   EXISTS (SELECT * FROM Role r
                         WHERE f.titre = r.titre
                         AND   EXISTS (SELECT * FROM Artiste a2
                                       WHERE r.id_acteur = a2.id
                                       AND   nom = 'Hackman')))
```

Je laisse le lecteur déchiffrer cette dernière requête (elle fonctionne !) et se convaincre que l'argument de lisibilité des requêtes imbriquées atteint rapidement ses limites. De plus ce genre d'expression sera probablement plus difficile à traiter pour le système.

En résumé, une jointure entre les tables *R* et *S* de la forme :

```
SELECT R.*
FROM R, S
WHERE R.a = S.b
```

peut s'écrire de manière équivalente avec une requête imbriquée :

```
SELECT *
FROM R
WHERE R.a IN (SELECT S.b FROM S)
```

ou bien encore sous forme de requête corrélée :

```
SELECT *
FROM R
WHERE EXISTS (SELECT S.b FROM S WHERE S.b = R.a)
```

Le choix de la forme est matière de goût ou de lisibilité, ces deux critères relevant de considérations essentiellement subjectives.

10.4.3 Requêtes avec négation

Les requêtes imbriquées sont en revanche irremplaçables pour exprimer des *négations*. On utilise alors `NOT IN` ou (de manière équivalente) `NOT EXISTS`. Voici un premier exemple avec la requête : *donner les films pour lesquels on ne connaît aucun rôle*.

```
SELECT * FROM Film
WHERE  titre NOT IN (SELECT titre FROM Role);
```

On obtient le résultat suivant :

```
+-----------+-------+----------------+-------+
| titre     | annee | id_realisateur | genre |
+-----------+-------+----------------+-------+
| Kagemusha |  1980 |             68 | Drame |
+-----------+-------+----------------+-------+
```

La négation est aussi un moyen d'exprimer des requêtes courantes comme celle recherchant le (ou les) films le(s) plus ancien(s) de la base. En SQL, on utilisera typiquement une sous-requête pour prendre l'année minimale parmi les années de production des films, laquelle servira à sélectionner un ou plusieurs films.

```
SELECT *
FROM Film
WHERE annee =  (SELECT MIN(annee) FROM Film)
```

Il existe en fait beaucoup de manières d'exprimer la même chose avec un SQL « complet ». Tout d'abord cette requête peut en fait s'exprimer sans la fonction `MIN()`, avec la négation : si *f* est le film le plus ancien, c'est en effet *qu'il n'existe pas* de film strictement plus ancien que *f*. On utilise alors habituellement une requête dite « corrélée » dans laquelle la sous-requête est basée sur une ou plusieurs valeurs issues des tables de la requête principale.

```
SELECT *
FROM Film f1
WHERE NOT EXISTS  (SELECT annee FROM Film f2
                   WHERE f1.annee > f2.annee)
```

Le `f1.annee` dans la requête imbriquée appartient à la table référencée dans le `FROM` de la requête principale. Autre manière d'exprimer la même chose : si un film est le plus ancien, tous les autres sont plus récents. On peut utiliser le mot-clé `ALL`, qui indique que la comparaison est vraie avec *tous* les éléments de l'ensemble constitué par la sous-requête.

```
SELECT *
FROM Film
WHERE annee <= ALL (SELECT annee FROM Film)
```

On préfère en général `NOT EXISTS` à `ALL`, mais les deux sont équivalents, puisque quand une propriété est vraie pour *tous* les éléments d'un ensemble, il *n'existe pas* d'élément pour lequel elle est fausse. Dernier exemple de négation : quels artistes ne sont pas metteur en scène ? Les deux formulations ci-dessous sont équivalentes, l'une s'appuyant sur `NOT IN` et l'autre sur `NOT EXISTS`.

```
SELECT *
FROM Artiste
WHERE id NOT IN (SELECT id_realisateur FROM Film)

SELECT *
FROM Artiste
WHERE NOT EXISTS (SELECT * FROM Film WHERE Artiste.id = Film.id_realisateur)
```

Dans les deux cas, on trouve le résultat suivant :

```
+----+---------+---------+-----------------+
| id | nom     | prenom  | annee_naissance |
+----+---------+---------+-----------------+
| 21 | Hackman | Gene    |            1930 |
| 30 | Dutronc | Jacques |            NULL |
+----+---------+---------+-----------------+
```

10.5 AGRÉGATION

L'agrégation de données avec SQL a été expliquée de manière détaillée au moment de la présentation des algorithmes de prédiction, page 307, et nous n'y revenons que de manière assez brève. La syntaxe SQL fournit donc :

1. le moyen de partitionner une table en *groupes* selon certains critères ;
2. le moyen d'exprimer des conditions sur ces groupes ;
3. des fonctions d'agrégation.

Il existe un groupe par défaut : c'est la table toute entière. Sans même définir de groupe, on peut utiliser les fonctions d'agrégation.

```
mysql> SELECT COUNT(*), COUNT(nom), COUNT(annee_naissance)
    -> FROM Artiste;
+----------+------------+------------------------+
| COUNT(*) | COUNT(nom) | COUNT(annee_naissance) |
+----------+------------+------------------------+
|        5 |          5 |                      4 |
+----------+------------+------------------------+
```

On obtient 4 pour le nombre d'années, et 5 pour les autres valeurs. En effet l'attribut `annee_naissance` est `NULL` pour Jacques Dutronc, et n'est pas pris en compte par la fonction d'agrégation. Pour compter toutes les lignes, on doit utiliser `COUNT(*)`, ou un attribut déclaré comme `NOT NULL` : c'est le cas pour `COUNT(nom)`. On peut aussi compter le nombre de valeurs distinctes dans un groupe avec `COUNT(DISTINCT` *`expression`*`)`.

10.5.1 La clause `GROUP BY`

Pour bien analyser ce qui se passe pendant une requête avec `GROUP BY`, il faut décomposer l'exécution de la requête en deux étapes. Prenons l'exemple des films, groupés par genre.

```
SELECT  genre, COUNT(*), MAX(annee)
FROM    Film
GROUP BY genre
```

Dans une première étape, MySQL va constituer les groupes. On peut les représenter avec un tableau, comprenant pour chaque ligne des valeurs du (ou des) attribut(s) de classement (ici `genre`), associé(s) à toutes les lignes correspondant à cette valeur.

Le groupe associé au genre « Drame » est constitué de deux films : ce tableau n'est donc pas une table relationnelle, dans laquelle chaque cellule ne peut contenir qu'une seule valeur.

Pour se ramener à une table relationnelle, on transforme durant la deuxième étape chaque groupe de lignes en une valeur par application d'une fonction d'agrégation.

genre	titre, année, id_realisateur
Western	Impitoyable, 1992, 20
Drame	Van Gogh, 1990, 29
	Kagemusha, 1980, 68
Policier	Les pleins pouvoirs, 1997, 20

Les films groupés par genre

La fonction `COUNT()` compte le nombre de lignes dans chaque groupe, `MAX()` donne la valeur maximale d'un attribut parmi l'ensemble des lignes du groupe, etc. Nous rappelons la liste des fonctions d'agrégation dans le tableau 10.1.

Tableau 10.1 — Les fonctions d'agrégation de MySQL

Fonction	Description
COUNT(*expression*)	Compte le nombre de lignes.
AVG(*expression*)	Calcule la moyenne de *expression*.
MIN(*expression*)	Calcule la valeur minimale de *expression*.
MAX(*expression*)	Calcule la valeur maximale de *expression*.
SUM(*expression*)	Calcule la somme de *expression*.
STD(*expression*)	Calcule l'écart-type de *expression*.

```
mysql> SELECT  genre, COUNT(*), MAX(annee)
    -> FROM    Film
    -> GROUP BY genre;
+----------+----------+------------+
| genre    | COUNT(*) | MAX(annee) |
+----------+----------+------------+
| Policier |        1 |       1997 |
| Drame    |        2 |       1990 |
| Western  |        1 |       1992 |
+----------+----------+------------+
```

Dans la norme SQL ANSI, l'utilisation de fonctions d'agrégation pour les attributs qui n'apparaissent pas dans le `GROUP BY` est *obligatoire*. Une requête comme :

```
SELECT  genre, titre, COUNT(*), MAX(annee)
FROM    Film
GROUP BY genre
```

devrait être rejetée parce que le groupe associé à 'Drame' contient deux titres différents, et qu'il n'y a pas de raison d'afficher l'un plutôt que l'autre. MySQL est plus conciliant, et affiche une des valeurs trouvées :

```
mysql> SELECT  genre, titre, COUNT(*), MAX(annee)
    -> FROM    Film
    -> GROUP BY genre;
```

```
+----------+--------------------+----------+------------+
| genre    | titre              | COUNT(*) | MAX(annee) |
+----------+--------------------+----------+------------+
| Drame    | Van Gogh           |        2 |       1990 |
| Policier | Les pleins pouvoirs |        1 |       1997 |
| Western  | Impitoyable        |        1 |       1992 |
+----------+--------------------+----------+------------+
```

On a donc associé « Van Gogh » à « Drame », en oubliant 'Kagemusha'. Outre que cela ne présente pas beaucoup de signification, cette requête serait refusée par tout autre SGBD relationnel.

10.5.2 La clause HAVING

Finalement, on peut faire porter des conditions sur les groupes – ou plus précisément sur le résultat de fonctions d'agrégation appliquées à des groupes – avec la clause `HAVING`. La clause `WHERE` ne peut exprimer des conditions que sur les lignes prises une à une. Par exemple, on peut sélectionner les genres pour lesquels on connaît au moins deux films.

```
mysql> SELECT   genre, MAX(annee)
    -> FROM     Film
    -> GROUP BY genre
    -> HAVING   COUNT(*) >= 2;
+-------+------------+
| genre | MAX(annee) |
+-------+------------+
| Drame |       1990 |
+-------+------------+
```

La condition porte ici sur une propriété de l'ensemble des lignes du groupe, pas de chaque ligne prise individuellement. La clause `HAVING` est donc toujours exprimée sur le résultat de fonctions d'agrégation.

Pour conclure, voici une requête sélectionnant les metteurs en scène pour lesquels on ne connaît pas plus de deux films, avec le nombre de films, et un tri sur le nom du metteur en scène.

```
mysql> SELECT nom AS nom_realisateur, COUNT(f.titre) AS nbFilms
    -> FROM Film f, Artiste a
    -> WHERE  a.id= f.id_realisateur
    -> GROUP BY nom
    -> HAVING COUNT(*) <= 2
    -> ORDER BY nom;
+-----------------+---------+
| nom_realisateur | nbFilms |
+-----------------+---------+
| Eastwood        |       2 |
| Kurosawa        |       1 |
| Pialat          |       1 |
+-----------------+---------+
```

10.6 MISES À JOUR

Les commandes de mise à jour (insertion, destruction, modification) sont considérablement plus simples que les requêtes.

10.6.1 Insertion

L'insertion s'effectue avec la commande `INSERT`, avec trois variantes. Dans la première, on indique la liste des valeurs à insérer, sans donner explicitement le nom des attributs. MySQL suppose alors qu'il y a autant de valeurs que d'attributs, et que l'ordre des valeurs correspond à celui des attributs dans la table. On peut indiquer `NULL` pour les valeurs inconnues.

```
INSERT INTO Film
VALUES ('Vertigo', 1958, NULL, 'Suspense');
```

Si on veut insérer dans une partie seulement des attributs, il faut donner la liste explicitement.

```
INSERT INTO Film (titre, annee)
VALUES ('Vertigo', 1958);
```

Il est d'ailleurs préférable de toujours donner la liste des attributs. La description d'une table peut changer, par ajout d'attribut, et l'ordre `INSERT` qui fonctionnait un jour ne fonctionnera plus le lendemain.

Enfin, avec la troisième forme de `INSERT`, il est possible d'insérer dans une table le résultat d'une requête. Dans ce cas la partie `VALUES` est remplacée par la requête elle-même.

Par exemple on peut créer une table *ExtraitFilm* avec le titre du film, l'année, et le nom du metteur en scène, puis copier les informations de la base dans cette table avec les commandes combinées `INSERT ... SELECT`.

```
mysql> CREATE TABLE ExtraitFilm (titre VARCHAR(50) NOT NULL,
    ->                           annee INTEGER,
    ->                           nom_realisateur VARCHAR(30));
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO ExtraitFilm
    -> SELECT titre, annee, nom
    -> FROM   Film, Artiste
    -> WHERE  id_realisateur=id;
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM ExtraitFilm;
+----------------------------+------------------+
| titre                | annee | nom_realisateur   |
+----------------------+-------+------------------+
| Impitoyable          |  1992 | Eastwood          |
| Les pleins pouvoirs |  1997 | Eastwood          |
| Van Gogh             |  1990 | Pialat            |
| Kagemusha            |  1980 | Kurosawa          |
+----------------------+-------+------------------+
```

10.6.2 Destruction

La destruction s'effectue avec la clause `DELETE` dont la syntaxe est :

```
DELETE FROM table
WHERE condition
```

`table` étant bien entendu le nom de la table, et *`condition`* est toute condition, ou liste de conditions, valide pour une clause `WHERE`. En d'autres termes, si l'on effectue, avant la destruction, la requête

```
SELECT * FROM table
WHERE condition
```

on obtient l'ensemble des lignes qui seront détruites par `DELETE`. Procéder de cette manière est un des moyens de s'assurer que l'on va bien détruire ce que l'on souhaite.

10.6.3 Modification

La modification s'effectue avec la clause `UPDATE`. La syntaxe est proche de celle du `DELETE` :

```
UPDATE table SET A1=v1 , A2=v2 , ... An=vn
WHERE condition
```

Comme précédemment, `table` dénote la table, les A_i sont les attributs, les v_i les nouvelles valeurs et *`condition`* est toute condition valide pour la clause `WHERE`.

11

Récapitulatif PHP

Ce chapitre présente la syntaxe du langage PHP. Il est destiné à servir de référence, et non de tutoriel permettant d'apprendre le langage. Les exemples de code PHP ne manquent pas dans les chapitres qui précèdent, aussi nous restons assez concis dans ceux qui sont présentés ici.

PHP s'appuie sur toutes les structures et concepts d'un langage de programmation de haut niveau, ainsi – à partir de la version 5 – que sur les principales structures de la programmation orientée-objet. Il présente cependant la particularité d'avoir été conçu spécifiquement pour être intégré avec HTML.

L'intérêt de PHP réside également dans le très vaste ensemble de fonctions qui accompagnent le langage et fournissent une nombre impressionnant de fonctionnalités prêtes à l'emploi. L'annexe C donne une sélection de ces fonctions.

11.1 GÉNÉRALITÉS

Tout code PHP doit être inclus dans une balise <?php ... ?>. Des balises « courtes » <?> sont acceptées dans certaines configurations, mais elles ne sont pas recommandées.

Comme en C, le séparateur d'instructions est le point-virgule « ; ». Noter qu'une instruction « vide », marquée par un point-virgule est acceptée. La syntaxe suivante est donc correcte, bien que le second « ; » ne serve à rien.

```
echo "Ceci est une instruction"; ;
```

11.1.1 Commentaires

Il existe trois manières d'inclure des commentaires au sein du code PHP :

1. comme en C, entre les signes « /* » et « */ » ;
2. comme en C++, en commençant une ligne par « // » :
3. comme en *shell* Unix, avec « # ».

Les commentaires C++ ou Unix s'appliquent uniquement au texte qui suit sur la même ligne. La première méthode est donc la plus économique puisqu'on peut placer un texte de longueur arbitraire entre les deux symboles délimiteurs. Attention cependant à ne pas imbriquer des commentaires ! La deuxième ligne dans l'exemple ci-dessous fait partie du commentaire.

```
/* Je veux mettre un commentaire ..
 /* Je crois commencer un deuxième commentaire !
 */ Je crois fermer le deuxième ...
*/ Je crois fermer le premier, mais PHP ne comprend plus rien.
```

On peut mixer les styles de commentaires dans un même script.

11.1.2 Variables et littéraux

Les variables sont des symboles qui de référencent des valeurs. Comme leur nom l'indique, les variables peuvent référencer des valeurs différentes au cours de l'exécution d'un script, ce qui les distingue des *littéraux* ('0', '1.233', 'Ceci est une chaîne') qui représentent directement une valeur immuable.

> **REMARQUE –** PHP distingue les majuscules et minuscules dans le nom des variables : `$mavariable` et `$maVariable` désignent donc deux variables différentes. En revanche, les noms de fonctions sont insensibles à la casse.

Un nom de variable commence toujours par un '$', suivi d'au moins un caractère non-numérique (le '_' est autorisé), puis de n'importe quelle combinaison de chiffres et de caractères. Il est recommandé de se fixer une norme pour nommer les variables, et d'utiliser pour chacune un nom qui soit explicite de l'utilisation de cette variable.

Types

Le type de la valeur associée à une variable peut lui-même changer, contrairement à des langages typés comme le C. Au cours de la vie d'un script, une variable peut donc référencer un entier, puis une chaîne, puis un objet. L'interpréteur se charge de gérer l'espace mémoire nécessaire pour stocker la valeur référencée par une variable, ce qui représente un *très* grand confort d'utilisation par rapport au C qui doit explicitement allouer et désallouer la mémoire.

Déclaration

Il n'y a pas de déclaration de variable en PHP ! L'interpréteur crée automatiquement une variable dès qu'un nouveau symbole préfixé par '$' apparaît dans un script. L'instruction ci-dessous affecte la valeur 1 à la variable `$maVariable`, si elle n'existait pas avant.

```
$maVariable = 1;
```

Que se passe-t-il quand on utilise la valeur d'une variable qui n'a pas encore été définie ? Et bien PHP rencontre un nouveau symbole préfixé par '$' et instancie donc une nouvelle variable avec comme valeur initiale la chaîne vide ou 0, selon le contexte. Le code suivant, où on a fait une faute de frappe dans le nom de la variable, affichera ainsi un blanc.

```
echo "$maVaraible";
```

PHP passe silencieusement sur ce genre d'erreur, sauf si on a défini un niveau d'erreur suffisamment restrictif (voir page 222). Il est fortement recommandé, au moins en phase de développement, d'adopter un niveau `E_ALL` pour détecter ce genre d'anomalies. La fonction `isSet()` est également très utile pour savoir si une variable a été déjà définie ou non.

Variable de variable

Le nom d'une variable peut lui-même être une variable. Le code ci-dessous affecte la valeur 10 à la variable `$mavar`, dont le nom est lui-même la valeur de la variable `$v1`. Cette construction assez exotique a une utilité douteuse.

```
$v1 = "mavar";
$$v1 = 10;
```

11.1.3 Constantes

Une constante est un symbole associé à une valeur mais, à la différence des variables, ce symbole ne peut jamais être modifié. Une constante peut être vue comme un littéral désigné de manière symbolique, ce qui est préférable pour la clarté du code et son évolutivité. Les constantes sont définies par la commande `define`.

```
define("PI", 3.14116);
define ("MON_SERVEUR", "www.lamsade.dauphine.fr");
```

Par convention (mais ce n'est pas obligatoire) les constantes sont en majuscules. Une bonne pratique est de ne *jamais* utiliser de valeur « en dur » dans un script, mais de définir une constante. Deux avantages :

1. le code est plus lisible ;
2. si on veut changer la valeur, on peut le faire en un seul endroit.

11.2 TYPES

PHP distingue les *types scalaires* (entiers, flottants, chaînes) et les *types agrégats* (tableaux et classes). Les valeurs de types scalaires ne peuvent pas se décomposer, contrairement à celles des types agrégats.

11.2.1 Types numériques et booléens

Ils comprennent les entiers et les flottants, ces derniers ayant une partie décimale séparée de la partie entière par un « . ».

```
$i = 1;   // Entier en notation décimale
$i = 011; // Notation octale (9 en décimal)
$i = 0x11;// Notation hexadécimale (17 en décimal)
$f = 3.1416; Flottant
$f = 0.3e-3; Notation exponentielle (soit 0,0003)
```

PHP n'a pas de type booléen explicite. Comme en C, la valeur *faux* est le 0, la chaîne vide ou la chaîne "0", et toute autre valeur est *vraie*, y compris un nombre négatif par exemple. Les constantes `TRUE` et `FALSE` sont prédéfinies et peuvent être utilisées dans les structures de contrôle.

11.2.2 Chaînes de caractères

Les chaînes de caractères peuvent êtres encadrées par des apostrophes simples (') ou doubles ("). Les premières peuvent contenir des apostrophes doubles, et réciproquement. Les deux types de chaînes ne sont cependant pas équivalents.

Apostrophes simples

On ne peut y inclure ni variables, ni caractères d'échappement (comme `'\n'`). En revanche les sauts de lignes sont acceptés. Si on veut inclure un (') dans une telle chaîne, il faut le préfixer par '\'. Exemple :

```
'C\'est une chaîne avec apostrophes simples

 et un saut de ligne.'
```

Apostrophes doubles

Contrairement aux précédentes, ces chaînes peuvent inclure des noms de variables qui seront remplacées par leur valeur à l'exécution. Dans l'exemple ci-dessous, on obtiendra la phrase *Le réalisateur de Vertigo est Hitchcock*. Ce mécanisme est extrêmement utile dans un langage orienté vers la production de texte.

```
$nom = "Hitchcock";
echo "Le réalisateur de Vertigo est $nom.";
```

On peut aussi utiliser les caractères d'échappements donnés ci-dessous.

Caractère	Description
\n	Saut de ligne
\r	Retour chariot
\t	Tabulation
\\	Le signe '\'
\$	Le signe '$'
\"	Une apostrophe double
\0nn	Une chaîne en octal
\xnn	Une chaîne en héxadécimal

Quand on veut insérer dans une chaîne de caractères un élément d'un tableau ou une propriété d'un objet, on peut les encadrer par des accolades ({}) pour que l'interpréteur puisse les distinguer.

```
echo "Le second élément du tableau 'tab': {$tab[2]}";
echo "La propriété 'val' de l'objet 'o': {$o->val}";
```

PHP tolère l'absence des accolades pour les objets.

11.2.3 Tableaux

Un tableau est une suite de valeurs référencées par une unique variable. PHP gère dynamiquement la taille des tableaux, ce qui permet d'ajouter ou de supprimer à volonté des valeurs sans se soucier de l'espace nécessaire.

Les tableaux en PHP peuvent être soit *indicés* – les valeurs sont référencées par leur position en débutant à 0 – soit *associatifs*. Dans ce cas les valeurs sont référencées par des noms, ou *clés*, donnés explicitement par le programmeur.

Tableaux indicés

Voici quelques exemples de tableaux indicés.

```
$tab[0] = "élément 1 ";
$tab[1] = "élément 2 ";
$tab[2] = 120;
```

Comme le montre cet exemple, on peut mêler des chaînes de caractères et des numériques dans un tableau. Notez bien que les indices commencent à 0, ce qui nécessite parfois un peu de réflexion quand on programme des itérations sur un tableau.

Une caractéristique importante et très utile de PHP est l'affectation automatique d'un indice à un nouvel élément du tableau. Cet indice est le numéro de la première cellule vide. Donc le code ci-dessous est équivalent au précédent.

```
$tab[] = "élément 1 "; // $tab[0] !
$tab[] = "élément 2 "; // $tab[1] !
$tab[] = 120; // $tab[2] !
```

L'instruction `array` offre un moyen d'initialiser facilement un tableau. Encore une fois, PHP indice respectivement par 0, 1 et 2 les éléments du tableau.

```
$tab = array ( "élément 1 ", "élément 2 ", 120);
```

Tableaux associatifs

L'utilisation d'un indice numérique pour désigner les éléments d'un tableau peut être généralisée en utilisant des chaînes de caractères ou *clés*. La clé d'un élément doit être unique pour l'ensemble du tableau. Dans ce cas, la notion d'ordre disparaît et on obtient une structure de *tableau associatif* qui, comme son nom l'indique, permet d'accéder à un élément par sa clé. Un tableau indicé est un cas particulier de tableau associatif, où les clés sont des entiers en séquence.

Voici l'initialisation d'un tableau `$mes` qui associe à un titre de film, utilisé comme clé, le nom de son metteur en scène.

```
$mes["Vertigo"] = "Hitchcock";
$mes["Sacrifice"] = "Tarkovski";
$mes["Alien"] = "Scott";
```

Comme précédemment, on peut utiliser le mot-clé `array` pour initialiser ce tableau, avec une syntaxe un peu plus complexe qui permet de donner la clé de chaque élément.

```
$mes = array (
  "Vertigo" => "Hitchcock",
  "Sacrifice" => "Tarkovski",
  "Alien" => "Scott");
```

Maintenant il est possible d'accéder à un élément du tableau avec sa clé. Par exemple `$mes["Vertigo"]` aura pour valeur `"Hitchcock"`. Le parcours d'un tableau associatif devient un peu plus complexe que celui des tableaux indicés puisqu'on ne peut pas se baser sur l'ordre des indices pour effectuer une boucle simple. On peut utiliser un *curseur* sur le tableau. Les fonctions `next()` et `prev()` permettent de déplacer ce curseur (initialement positionné sur le premier élément du tableau), et les fonctions `key()` et `current()` renvoient respectivement la clé et la valeur de l'élément courant. Voici un exemple de code pour parcourir le tableau `$mes`.

```
do
{ echo "Clé = key($mes). Valeur = current($mes)"}
while (next($mes));
```

En général, on préfère recourir à l'instruction `foreach`, comme le montre l'exemple ci-dessous.

```
foreach ($mes as $cle => $valeur)
{
  echo  "Clé = $cle. Valeur = $valeur";
}
```

La fonction `count()` donne le nombre d'éléments dans un tableau, et il existe tout un ensemble de fonctions pour trier un tableau associatif selon différents critères : `sort()`, `asort()`, `arsort()`, `ksort()`, etc. Nous vous renvoyons à l'annexe C pour les fonctions PHP manipulant des tableaux.

Tableaux multi-dimensionnels

Les tableaux indicés et associatifs se généralisent aux tableaux multi-dimensionnels, pour lesquels l'indice, ou la clé, est constituée de plusieurs valeurs. Un tableau à deux dimensions peut être vu comme une table avec lignes et colonnes. Voici par exemple un tableau avec deux lignes et deux colonnes.

```
$tab[0][0] = "En haut à gauche";
$tab[0][1] = "En haut à droite";
$tab[1][0] = "En bas à gauche";
$tab[1][1] = "En bas à droite";
```

On peut utiliser des indices ou des clés, et même mixer les deux. Quant au constructeur `array`, il peut être imbriqué pour initialiser des tableaux multi-dimensionnels.

```
$mes = array (
  "Vertigo" => array ( "Alfred", "Hitchcock"),
  "Sacrifice" => array ( "Andrei", "Tarkovski"),
  "Alien" => array ( "Ridley", "Scott"));
```

Dans l'exemple ci-dessus, les tableaux imbriqués sont indicés et contiennent chacun deux éléments. `$mes["Vertigo"][1]` est donc la chaîne `"Hitchcock"`.

11.2.4 Conversion et typage

Le type d'une variable est déterminé par le contexte dans lequel elle est utilisée. Quand on implique par exemple une chaîne de caractères dans une addition, PHP essaiera d'en extraire un numérique.

```
$r = 1 + "3 petits cochons";
```

Le code précédent affecte la valeur 4 à `$r` puisque la chaîne est convertie en 3. Si la conversion s'avère impossible, la valeur 0 est utilisée, mais dans ce cas votre script souffre d'un sérieux défaut. Ce type de manœuvre est sans doute à proscrire dans tous les cas.

Typage

Il est possible de tester le type d'une variable (ou, plus précisément, le type de la valeur référencée par la variable) avec les fonctions `is_long()` (pour un entier), `is_double()` (pour un flottant), `is_string()`, `is_array()` et `is_object()`. Ces fonctions booléennes renvoient `TRUE` ou `FALSE`.

Autre possibilité : la fonction `getType()`, appliquée, à une variable, renvoie `"integer"`, `"double"`, `"string"`, `"array"` ou `"object"`.

Conversion

On peut convertir le type d'une variable – ou, plus précisément, de la valeur d'une variable ...– en préfixant le nom de la variable par `(type)` où `type` est `integer`, `double`, `string`, etc.

```
$v = "3 petits cochons";
$v = (integer) $v; // Maintenant $v vaut 3
$v = (double) $v; // Maintenant $v vaut 3.0
```

11.3 EXPRESSIONS

On désigne par *expression* toute construction du langage qui produit une valeur. Les variables, littéraux et constantes sont déjà des expressions : elles produisent leur propre valeur.

Le code ci-dessous contient 3 expressions. La première instruction, 10, a pour valeur 10 ! L'affectation qui suit se base sur le fait que la syntaxe "10" produit la valeur 10, et donne donc à la variable `$i` elle-même la valeur 10. Enfin la dernière instruction produit, elle aussi, 10, qui est la valeur de la variable.

```
10;
$i = 10;
$i;
```

À peu près toutes les constructions syntaxiques ont une valeur en PHP, et sont donc des expressions. Revenons sur l'affectation de la valeur 10 à la variable `$i` : cette affectation a elle aussi une valeur, qui est 10 ! On peut donc logiquement écrire.

```
$j = $i = 10;
```

La variable `$j` prend la valeur de l'affectation `$i = 10`, soit 10. Ce n'est pas forcément une bonne habitude de programmation que d'utiliser ce genre de construction qui peut être difficile à comprendre, mais cela illustre le principe de base du langage (le même qu'en C) : on manipule des valeurs produites par des expressions.

11.4 OPÉRATEURS

Après les variables et les littéraux, les *opérateurs* constituent le moyen le plus courant de créer des expressions. Un opérateur produit une valeur par manipulation –addition, soustraction, etc.– de valeurs fournies par d'autres expressions. Dans leur forme la plus simple, les opérateurs agissent sur des variables ou des littéraux.

```
$a = 3;
$a + 4;
$b = $a + 2;
```

Le code ci-dessus donne quelques exemples. La première ligne est une affectation qui, nous l'avons vu, produit la valeur de son opérande la plus à droite. La seconde ligne est une addition qui produit la valeur 7, ce qui ne sert pas à grand chose puisque cette valeur n'est stockée nulle part.

La troisième ligne est un peu plus complexe car elle combine addition et affectation. La partie droite effectue la somme de `$a`, dont la valeur est 3, avec le littéral 2 dont la valeur est 2. Cette expression produit la valeur 5 qui est affectée à `$b`. Enfin, la composition des deux expressions (addition et affectation) produit elle-même une valeur, 5.

On peut remarquer que toute expression peut être interprétée comme une *expression booléenne* avec la valeur `FALSE` si la valeur produite est égale à 0 (ou à une chaîne vide), `TRUE` sinon. Toute expression peut donc être utilisée dans les structures de test, ce que nous exploiterons plus loin.

Les opérateurs peuvent être *composés* pour former des expressions complexes.

```
$a = 3;
$b = 8;
$c = $a + 2 * $b;
```

Que vaut la dernière expression ? Est-ce 5 fois 8, soit 40, ou 3 + 16, soit 19 ? Le résultat est défini par *l'ordre de précédence* des opérateurs. Ici la multiplication a un ordre de précédence supérieur à l'addition, et sera donc évaluée d'abord, ce qui donne le second résultat, 19. Un langage de programmation se doit de définir précisément l'ordre de précédence de ses opérateurs, ce qui ne signifie pas (heureusement) que le programmeur doit les connaître. En utilisant des parenthèses, on fixe sans ambiguïté l'ordre d'exécution et on obtient un code bien plus facile à lire. Les expressions ci-dessus gagneront à être écrites comme :

```
$a = 3;
$b = 8;
$c = $a + (2 * $b);
```

Nous ne donnons pas l'ordre de précédence des opérateurs PHP, mais vous les trouverez dans la documentation si vous y tenez vraiment. De toute manière il est fortement recommandé d'utiliser les parenthèses pour rendre explicites les expressions arithmétiques. La table 11.1 donne la liste des opérateurs arithmétiques de PHP.

Tableau 11.1 — Opérateurs arithmétiques

Opérateur	Description
`$a + $b`	Addition de `$a` et `$b`
`$a - $b`	Soustraction de `$b` à `$a`
`$a * $b`	Multiplication de `$a` et `$b`
`$a / $b`	Division de `$a` par `$b`
`$a % $b`	`$a` modulo `$b` (reste de la division de `$a` par `$b`)

11.4.1 Concaténation de chaînes

Un opérateur très fréquemment utilisé est la *concaténation de chaînes*, simplement représentée par le point « . ».

```
$c1 = "Bonjour "';
$c2 = "Dominique";
// Affichage de la chaîne "Bonjour cher Dominique"
echo $c1 . " cher " . $c2;
```

Nous avons déjà discuté de l'opérateur d'affectation '=', qu'il ne faut surtout pas utiliser pour effectuer des tests d'égalité ! Il est couramment utilisé pour stocker dans une variable le résultat d'un calcul.

```
$i = 4 + 3;
$c = "Bonjour cher";
$i = $i + 2;
$c = $c . " Dominique"
```

Les deux dernières formes sont très courantes : elles utilisent une variable à la fois comme opérande (dans la partie droite de l'affectation) et pour stocker le résultat (dans la partie gauche de l'affectation). Il existe une syntaxe permettant d'exprimer de manière plus économique ce type d'opération.

```
$i += 2;
$c .= " Dominique"
```

Cette notation évite d'avoir à répéter le nom de la variable. Elle peut être utilisée avec tous les opérateurs arithmétiques et l'opérateur de concaténation.

11.4.2 Incrémentations

Une autre technique très couramment utilisée est l'incrémentation d'un compteur, dans une boucle par exemple. L'écriture normale est `$i = $i + 1`. PHP reprend du C une syntaxe plus compacte : l'expression `$i++` est équivalente à la précédente et incrémente de 1 la valeur de `$i`.

Comme toute expression, `$i++` a une valeur, en l'occurrence la valeur de `$i` *avant* son incrémentation. Pour obtenir la valeur `$i` *après* son incrémentation, on utilise

++$i. Les mêmes expressions existent avec l'opérateur de soustraction. Ces subtilités du langage sont liées à la volonté de faire de PHP un langage où tout (presque tout) est expression, c'est-à-dire a une valeur. On peut tout à fait s'épargner l'utilisation de ce genre de technique si on trouve qu'elles ne favorisent pas la qualité du code. Voici quelques exemples résumant ces opérateurs unaires.

```
$i = 4 ; // $i vaut 4
$i++; / $i vaut 5
$j = ++$i; // $j vaut 6, $i vaut 6
$k = $i++; // $k vaut 6, $i vaut 7
$k--; // $k vaut 5
```

11.4.3 Opérateurs de bits

La table 11.2 donne la liste des opérateurs de bits. Il agissent sur des entiers et permettent de manipuler la représentation binaire. En déplaçant tous les bits d'un cran vers la gauche par exemple, on obtient le double de sa valeur.

Tableau 11.2 — Opérateurs de bits

Opérateur	Description
`$a & $b`	*ET* binaire. Renvoie un entier dont les bits à 1 sont ceux à 1 dans $a ET dans $b.
`$a | $b`	*OU* binaire. Renvoie un entier dont les bits à 1 sont ceux à 1 dans $a OU dans $b.
`$a ^ $b`	*OU EXCLUSIF* binaire. Renvoie un entier dont les bits à 1 sont ceux à 1 dans $a OU dans $b, mais pas dans les deux.
`~ $a`	Renvoie un entier dont les bits sont inversés par rapport à ceux de la variable $a.
`$a << $b`	Décale les bits de $a de $b positions vers la gauche.
`$a >> $b`	Décale les bits de $a de $b positions vers la droite.

11.4.4 Opérateurs logiques

Les opérateurs logiques sont donnés dans la table 11.3. Rappelons qu'une valeur est interprétée comme vraie si elle est différente de 0 ou de la chaîne vide, et fausse sinon.

Tableau 11.3 — Opérateurs logiques

Opérateur	Description
`$a && $b`	(ET) Renvoie vrai si $a ET $b sont vrais.
`$a and $b`	Idem que &&.
`$a || $b`	(OU) Renvoie vrai si $a OU $b est vrai.
`$a or $b`	Idem que ||.
`$a xor $b`	Ou exclusif : $a OU $b est vrai, mais pas les deux.
`!$a`	(NOT). Renvoie la négation de $a.

Enfin la table 11.4 donne la liste des opérateurs de comparaison de PHP. Il est très important de noter, si vous n'êtes pas familier du langage C ou de ses dérivés (C++, Java) que le test d'égalité s'écrit avec deux '='. Une erreur très courante est d'oublier un '=' dans un test d'égalité. L'interpréteur ne signale rien puisque l'opérateur d'affectation renvoie une valeur qui peut être interprétée comme un booléen.

```
$i = 1;
$j = 2;

if ($i == $j) ... // Renvoie FALSE: i est différent de j.
if ($i = $j) ... // Renvoie TRUE !
```

Dans l'exemple ci-dessus, le deuxième test utilise (par erreur) l'affectation '=' et non la comparaison '=='. Dans ce cas la valeur `$j` est affectée à `$i`, et la valeur de cette affectation est elle-même la valeur de `$j`, soit 2, interprétée comme `TRUE`. Non seulement le test ne donne pas le résultat attendu, mais la valeur de `$i` a été modifiée.

Ce genre d'erreur est difficile à détecter, et fait partie des faiblesses de la syntaxe du C qui a été reprise dans PHP.

Tableau 11.4 — Opérateurs de comparaison

Opérateur	Description
`$a == $b`	Vrai si $a est égal à $b.
`$a != $b`	Vrai si $a est différent de $b.
`$a < $b`	Vrai si $a est inférieur à $b.
`$a > $b`	Vrai si $a est supérieur à $b.
`$a <= $b`	Vrai si $a est inférieur ou égal à $b.
`$a >= $b`	Vrai si $a est supérieur ou égal à $b.

11.5 STRUCTURES DE CONTRÔLE

Les structures de contrôles sont les *tests* et les *boucles*. Elles permettent de spécifier, en fonction de l'état d'un script à l'exécution (déterminé par la valeur de certaines variables), quelles sont les parties du script à effectuer (structures de tests), ou combien de fois on doit les effectuer (structures de boucle).

L'unité de base pour décrire les actions d'un script est *l'instruction*, qui est typiquement l'affectation d'une valeur à une variable, ou toute autre expression plus complexe. Les instructions sont séparées par des points virgules. On place le contrôle sur des *blocs*. Un bloc est une suite d'instructions, exécutées en séquence et solidairement, et encadrées par '{' et '}'. Là encore cette syntaxe est reprise du C.

11.5.1 Tests

La structure la plus courante est le `if ... else`. Voici la syntaxe.

```
if (expression)
{
 // Bloc si expression est vraie.
}
else
{
 // Bloc si expression est fausse.
}
// Ici le script continue.
```

Le `else`, et son bloc associé, est optionnel. De même, si le bloc du `if` contient une seule instruction, on peut se passer des accolades. *`expression`* est en général un test de comparaison, éventuellement plusieurs avec des opérateurs logiques. Comme nous l'avons vu, on peut utiliser n'importe quelle expression, y compris des appels de fonction, puisque toute valeur peut être interprétée comme *vrai* ou *faux*.

Les instructions `if` peuvent être imbriquées (un `if` dans un autre `if`) et cumulées (plusieurs `if` consécutifs), comme le montre l'exemple ci-dessous. Il n'y a pas de limite à la profondeur d'imbrication des `if`, si ce n'est le manque de lisibilité qui en découle. Une bonne règle est de ne pas s'autoriser plus de deux niveaux d'imbrications. Au-delà, le code est trop complexe et il vaut mieux recourir à des fonctions.

```
if (expr1)
{
 if (expr2)
 {
  // expr1 et expr2 sont vraies.
 }
 else
 {
  // expr1 vraie, expr2 fausse
 }
}
else if (expr3)
{
 // expr1 fausse, expr3 vraie.
}
else
{
 // expr1 et expr3 fausses.
}
```

La structure `else if` illustrée ci-dessus peut s'écrire également `elseif`. Elle permet notamment de passer en revue les différentes valeurs possibles pour une expression, et d'exécuter le bloc approprié. Une structure équivalente pour ce type de test est le `switch`.

```
switch (expression)
{
 case valeur1:
  // expression vaut valeur1.
  // Il faut sortir du switch !
  break;
 case valeur2:
  // expression vaut valeur2.
  break;
...
 default:
  // expression ne vaut aucune des valeurs listées
  break;
}
```

Le `switch` est une structure de test initialisée avec l'instruction `switch (expression)` qui définit *`expression`* comme la valeur que l'on va ensuite comparer avec une liste de possibilités. Chaque possibilité de cette liste est représentée par une instruction « `case valeur:` ». Si *`expression`* est égale à *`valeur`*, alors les instructions qui suivent sont exécutées, *jusqu'à la fin du* `switch`, *et ce indépendamment des* `case` ! Pour indiquer que l'on veut sortir de la structure `switch` dès que la valeur de *`expression`* a été rencontrée, et le bloc exécuté, il faut finir ce bloc par une instruction `break`, comme indiqué ci-dessus.

À la fin du bloc `switch`, il est possible d'ajouter le mot-clé `default` qui indique l'ensemble des instructions à exécuter si tous les `case` précédents ont échoué.

Signalons, pour conclure sur les tests, qu'il existe une syntaxe légèrement différente pour les `if-else`. Le bloc après le `if` ou le `else` commence par ':', et la structure se termine par `endif;`.

```
if (expression):
 // Bloc si expression est vraie.
else:
 // Bloc si expression est fausse.
endif;
```

11.5.2 Boucles

Comme d'habitude, on retrouve les même constructions qu'en C, ainsi qu'une instruction `foreach` pratique pour parcourir les tableaux associatifs.

Le `while`

Le `while` permet d'exécuter un bloc d'insructions tant qu'une condition est remplie. La structure est :

```
while (expression)
{
 // expression est vraie.
}
```

On ne rentre dans la boucle que si la condition est vraie au départ. Il va sans dire qu'il doit y avoir dans la boucle au moins une instruction qui fait varier la valeur de *expression* de manière à ce que celle-ci devienne fausse après un nombre fini d'itérations. Une syntaxe légèrement différente du `while` marque le bloc à exécuter de manière différente.

```
while (expression):
 // expression est vraie.
endwhile;
```

Le `do-while`

Le `do-while` est une variante du `while` qui effectue le bloc *avant* d'évaluer le test. En conséquence, ce bloc est toujours exécuté au moins une fois.

```
do
{
 // expression n'est pas forcément vraie au premier passage
}
while (expression);
```

Le `for`

L'instruction `for` permet d'exécuter une itération sur une variable incrémentée (ou décrémentée) à chaque passage de la boucle. C'est la plus puissante des structures de boucle puisqu'on peut y spécifier

- l'initialisation des valeurs conditionnant l'itération ;
- la ou les instructions faisant évoluer ces valeurs à chaque passage ;
- la condition d'arrêt de la boucle.

Par exemple, considérons la partie de code suivante :

```
$x = 0;
while ($x < 10)
{
 // Ici des instructions
 $x++; // permet d'incrémenter $x de 1. Équivalent à $x = $x+1
}
```

Par construction, on passera 10 fois dans la boucle. On peut écrire cela plus concisément avec `for` :

```
for ($x=0; $x < 10; $x++)
{
 // Ici des instructions
}
```

Dans le `for`, la première partie initialise la variable de contrôle (ici `$x`), la deuxième indique le test de fin, et la troisième effectue l'incrémentation. On peut

mettre des choses beaucoup plus compliquées :

```
$a=1;
$b=6;
while ($a < $b)
{
 $a++;
 echo "$a = " . $a);
}
```

peut être remplacé par :

```
for ($a=1,$b=6; $a < $b; $a++, echo "$a = " . $a);
```

On effectue donc une boucle `for` sur une instruction vide (';'), les actions étant intégrées à l'instruction `for` elle-même. Ce mode de programmation très compact est apprécié de certains programmeurs, en particulier de ceux qui se font une gloire de produire un code court et illisible.

Le `foreach`

L'instruction `foreach` est conçue pour parcourir facilement un tableau. Il en existe deux formes, selon que le tableau est indicé ou associatif.

```
foreach ($tableau as $valeur) { /* bloc */ }  // Tableau indicé
foreach ($tableau as $cle => $valeur) { /* bloc */ } // Tableau associatif
```

À chaque étape du parcours de `$tableau`, le curseur interne est incrémenté, et la valeur de l'élément courant (ainsi que la clé dans la seconde forme) est affectée à `$valeur` (ainsi qu'à `$cle`).

11.5.3 Les instructions `break` et `continue`

En principe, quand on est dans le corps d'une boucle, toutes les instructions seront exécutées jusqu'à ce que le test qui détermine si on doit effectuer ou non une nouvelle itération soit évalué. Il existe deux instructions qui permettent d'obtenir un comportement plus souple.

1. `break` déclenche la sortie forcée de la boucle ;
2. `continue` dirige l'exécution à la prochaine évaluation du test de continuation, en sautant les éventuelles instructions complétant le corps de la boucle.

Voici un exemple illustrant ces deux instructions. On effectue une boucle avec 10 itérations, mais au lieu de s'appuyer sur le mécanisme normal du `while`, on utilise `break` pour sortir de la boucle quand la variable de contrôle vaut 10.

```
$x = 0;
while (1)
{
 if ($x == 10) break; // $x vaut 10 ? On s'en va
```

```
 $x++; // incrémente $x de 1. Équivalent à $x = $x+1
 if ($x != 5) continue; // $x différent de 5 ? On saute la suite
 // Ici les instructions pour le cas où $x vaut 5
}
```

Noter le `while (1)` qui revient à effectuer une boucle à priori infinie. Le seul moyen de l'interrompre est alors le `break`, ou `exit` qui interrompt le script. L'instruction `continue` est déclenchée si la valeur de `$x` est différente de 5, ce qui revient à aller directement au `while` en ignorant la suite du corps de la boucle. Ces instructions sont valables également pour les variantes `for` et `do`.

11.6 FONCTIONS

Les fonctions en PHP doivent être définies avant leur appel. Le nom d'une fonction ne commence pas par '$', et n'est pas sensible à la casse. Voici la syntaxe de définition d'une fonction.

```
function NomFonction ([£arg1, £arg2, ...])
{
 // Ici le code de la fonction
}
```

Une fonction peut prendre un nombre arbitraire, mais *fixe* de variables. Les fonctions avec un nombre variable d'arguments n'existent pas en PHP, mais on on peut contourner cette limitation en passant un tableau en paramètre [1]. Une fonction peut renvoyer une valeur avec l'instruction `return`, mais ce n'est pas indiqué dans sa signature. La valeur renvoyée peut être un tableau.

```
function FilmVertigo ()
{
 return array ("Vertigo", "Alfred", "Hitchcock");
}

// Appel de la fonction
list ($titre, $prenom, $nom) = FilmVertigo ();
```

11.6.1 Passage des arguments

Les arguments sont passés *par valeur*, ce qui signifie qu'une copie des variables est faite au moment de l'appel de la fonction, et que les éventuelles modifications faites dans le corps de la fonction sur les arguments n'ont qu'un effet local. Il existe une exception depuis la version 5 : les objets sont passés par référence.

1. Il existe cependant des techniques avancées pour gérer des listes de paramètres quelconques : voir dans la documentation PHP les fonctions `func_num_args()`, `func_get_args()` et `func_get_arg()` si cela vous intéresse.

```
function Addition ($i, $j)
{
 // NB : $i et $j sont des variables locales
 $somme = $i + $j;
 $i = 2; $j = 3;
 return $somme;
}

$i = 1; $j = 1;
// Appel de la fonction
$k = Addition ($i, $j);
// $i et $j valent toujours 1 !
```

Le passage par valeur est préférable, parce qu'il signifie que toutes les actions de la fonction n'ont pas d'impact sur le script qui l'appelle. En conséquence, le script n'a pas à savoir comment la fonction est implantée et n'a pas à craindre des « effets de bord » indésirables.

Il est cependant possible de passer des variables *par référence*. C'est alors l'adresse de la variable qui est transmise à la fonction, et pas une copie de sa valeur. Il existe deux manières d'indiquer un passage par référence :

1. au moment de l'appel, même pour une fonction qui a été prévue pour travailler sur des arguments passés par valeur ;
2. dans la définition de la fonction. C'est alors la règle par défaut pour tout appel de la fonction.

La première option est très dangereuse et va probablement disparaître dans une prochaine version de PHP. Il faut l'éviter. La seconde méthode, avec des arguments explicitement prévus pour être passés par référence, rejoint une technique classique, et présente moins de dangers. Voici une version de la fonction `Addition()` qui prend en argument une troisième variable, passée par référence afin de stocker le résultat en sortie.

```
function Addition ($i, $j, &$somme)
{
 $somme = $i + $j;
 $i = 2; $j = 3;
}

$i = 1; $j = 1;
// Appel de la fonction : $k est passée par adresse
Addition ($i, $j, $k);
// $i et $j valent toujours 1, mais $k vaut 2 !
```

Un défaut de la syntaxe de PHP est qu'on ne peut pas savoir, en lisant un script, si les variables sont passées par valeur ou par adresse. Il est tout à fait possible de tout écrire sans recourir au passage par adresse, ce qui élimine toute ambiguïté. C'est le choix de tous les scripts présentés dans ce livre. Le seul inconvénient potentiel est un éventuel problème de performance si on passe des variables volumineuses (commme un gros tableau) par valeur : voir la discussion page 61 à ce sujet.

11.6.2 Valeurs par défaut

Il est possible de définir des valeurs par défaut pour un ou plusieurs arguments d'une fonction. Cette idée, reprise du C++, s'avère très pratique quand on a beaucoup de paramètres dont la plupart prennent toujours la même valeur.

Voici l'exemple d'une fonction de connexion, dont on suppose que dans la plupart des cas elle sera utilisée pour accéder à la base *Films* sur le serveur *MonServeur*. On précise donc ces valeurs par défaut.

```
function Connexion ($pNom, $pMotPasse, $pBase = "Films",
       $pServeur = "MonServeur")
{
 // Ici le corps de la fonction
}
```

Maintenant on peut éventuellement omettre les deux derniers paramètres en appelant la fonction. On peut aussi les donner, et dans ce cas on se ramène à un passage classique de paramètres.

```
// Connexion à la base Films, sur le serveur MonServeur
$connexion1 = Connexion ("rigaux", "mdprigaux");
...
// Connexion à une autre base, sur un autre serveur.
$connexion2 = Connexion ("davy", "mdpdavy", "Films", "cartier");
```

On ne peut utiliser les valeurs par défaut que de droite à gauche dans la liste des arguments, et en commençant par le dernier. Il n'est pas possible par exemple, dans la fonction `Connexion()`, de donner une valeur par défaut au nom de la base et pas au serveur, à moins de modifier l'ordre des arguments. L'interpréteur doit toujours être capable de donner une valeur aux arguments non spécifiés au moment de l'appel. Le mot-clé `unset` peut être utilisé comme valeur par défaut. Dans ce cas l'argument correspondant n'est pas défini dans la fonction si cet argument n'est pas spécifié dans l'appel de la fonction.

11.6.3 Fonctions et variables

Quand on programme, on manipule des variables et des fonctions. Quels sont les rapports entre les deux ? Où doit-on déclarer une variable ? Quel est son degré de visibilité par rapport aux fonctions et/ou aux scripts ? PHP propose trois types de variables (nous reprenons la terminologie du C).

Variables automatiques. Ces variables sont créées dès que l'on entre dans leur espace de définition, qui est soit le script, soit une fonction¿; elles disparaissent quand on sort de cet espace.

Variables statiques. Une variable automatique est détruite quand on quitte le corps d'une fonction, et sera recréée au prochain appel de la fonction. Sa valeur

n'est donc pas sauvegardée entre deux appels. Les variables statiques, elles, sont persistantes entre deux appels.

Variables globales. En principe, le corps d'une fonction est une partie de code complètement isolée. En particulier, les variables définies à l'extérieur de la fonction ne sont pas visibles. Une variable globale est au contraire visible partout.

Par défaut les variables sont automatiques. Les variables statiques peuvent être utiles, mais on fait beaucoup mieux avec la programmation objet. Quant aux variables globales, on ne saurait trop déconseiller de les utiliser !

Variables automatiques

Quand une variable est déclarée *à l'intérieur* d'une fonction (c'est-à-dire entre les {} qui définissent le corps de la fonction), elle n'est pas visible – autrement dit, on ne peut pas lire ni modifier sa valeur – de l'extérieur de la fonction. Par exemple, la variable `nom` est uniquement accessible aux instructions de la fonction `MaFonc()` dans l'exemple ci-dessous :

```
MaFonc (...)
{
 $nom = "Vertigo";
...
}
```

Le terme « automatique » vient de l'allocation automatique de l'espace nécessaire au stockage de la variable, chaque fois que la fonction est appelée. Conséquence très importante : *le contenu de la variable automatique est perdu entre deux appels à une fonction*. Ceci est tout à fait cohérent avec l'idée qu'une fonction effectue une tâche bien précise, utilisant pour cela des variables temporaires de travail.

Variables statiques

Il existe un second type de variable dite *statique*, qui présente les propriétés suivantes :

1. elle n'est pas visible à l'extérieur de la fonction où elle est déclarée ;
2. entre deux appels à une fonction, une variable statique conserve sa valeur (le terme « statique » vient du C, et signifie que l'emplacement en mémoire de la variable est constant).

On peut par exemple définir une variable qui compte le nombre d'appels à une fonction.

```
MaFonc (...)
{
 // On définit la variable statique, et on l'initialise
 static $compteur = 0;
 ...
```

```
 // On incrémente la variable
 $compteur++;
 echo "Nombre d'appels à MaFonc : " . $compteur;
}
```

Les variables statiques offrent moins de possibilités en PHP qu'en C. Elles sont souvent avantageusement remplacées par la programmation objet.

Variables globales

Par opposition aux variables automatiques, une variable globale est définie à l'extérieur de toute fonction, et rendue accessible à l'intérieur des fonctions grâce au mot-clé `global`.

```
MaFonc ()
{
 // On définit la variable globale
 global $nom ;
 ....
 $nom = "N'importe quoi";
}
... // Beaucoup de lignes de codes.
$nom = "Vertigo";
MaFonc();
// La variable $nom contient "N'importe quoi" !!!
```

La variable `$nom` peut être manipulée, de manière totalement transparente, par la fonction `MaFonc()`. *Rien* n'indique que la fonction modifie la variable (elle n'est même pas passée en argument). En pratique, l'utilisation des variables globales est à proscrire. Les inconvénients sont innombrables.

1. *Manque de lisibilité* : quand on voit un appel de fonction dans un programme, on pense naturellement que la fonction ne manipule que les données passées en paramètre. C'est faux si on utilise des variables globales. D'où une grande difficulté à comprendre le code.
2. *Manque de sécurité* : si vous définissez une variable globale, tout le monde peut y toucher ! Donc impossible de faire des contrôles sur le contenu de cette variable. Impossible même de garantir qu'une fonction anodine n'est pas en train de la modifier quand vous exécutez un programme qui a l'air sain. La variable globale, c'est l'effet de bord institutionalisé.
3. *Manque d'évolutivité* : on fige pour toujours le nom d'une information, ce qui est mauvais pour l'évolution d'un programme.

Arrêtons là : on peut écrire une application de taille quelconque sans jamais utiliser de variable globale. Celà revient à donner la priorité aux fonctions sur les variables, et c'est un excellent principe.

11.7 PROGRAMMATION ORIENTÉE-OBJET

La programmation orientée-objet a considérablement évolué en PHP 5. Le chapitre 3 lui est entièrement consacré. Ce qui suit n'est donc qu'un rappel concis, complété des quelques fonctionnalités objets secondaires non évoquées dans le chapitre 3.

Une classe est un ensemble d'attributs et de fonctions (le terme technique est *méthodes*) manipulant ces attributs. Une méthode n'est rien d'autre qu'une fonction s'exécutant au sein d'un objet, dont l'environnement est constitué de l'ensemble des attributs – ou *propriétés*, ou encore *variables d'état* – constituant l'objet. Ces attributs sont accessibles au sein d'une méthode *via* la variable `$this`, qui désigne l'objet courant. La méthode accède à cet environnement, à ses propres paramètres, et à rien d'autre.

11.7.1 Classes et objets

Les définitions des attributs et des méthodes sont rassemblées dans la structure `class`. Voici l'esquisse d'une définition de classe pour gérer des objets géométriques, avec deux méthodes. L'une, `afficher()`, pour afficher l'objet sur un écran, l'autre, `surface()`, pour calculer la surface de l'objet.

Exemple 11.1 *exemples/Geom.class.php* : *une classe d'objets géométriques*

```
<?php
class Geom
{
  // ----   Partie privée : les propriétés
  private $abcisses, $ordonnees;  // Les coordonnées.
  private $nbPoints; // Nombre de points
  const PI = 3.14116;

  // Puis les méthodes
   public function ajoutPoint ($x, $y)
  {
   $this->abscisses[$this->nbPoints] = $x;
   $this->ordonnees[$this->nbPoints] = $y;
   $this->nbPoints++;
  }

  public function afficher ($ecran)
  {
   for ($i = 0; $i $<$ $this->nbPoints; $i++)
    $ecran->affiche($this->abcisses[$i], $this->ordonnees[$i]);
  }

  public function surface ()
  {
   // Ici un calcul de surface ...
   return $surface;
  }
}
```

Les propriétés et fonctions peuvent être qualifiées par `public`, `private` ou `protected`. Tout ce qui est `public` est accessible librement par les instructions manipulant un objet, alors que tout ce qui est `private` n'est accessible que par les méthodes de la classe. Le mot-clé `protected` indique que seules les sous-classes peuvent accéder à la propriété ou à l'attribut (voir plus loin).

On peut également associer des *constantes* à une classe, comme ici la constante `PI` avec sa valeur (qui ne change jamais !). On peut faire référence à cette constante interne avec la syntaxe `Geom::PI` (les constantes ne sont jamais privées en PHP).

Une classe est une sorte de moule pour construire – on parle *d'instanciation* – des objets se conformant à sa définition. On utilise le constructeur `new`.

```
$icone = new Geom;
// Ajout de quelques points
$icone->ajoutPoint(12, 43);
$icone->ajoutPoint(32, 30);
$icone->ajoutPoint(56, 6);
// Calcul de la surface
echo "Surface = " . $icone->surface();
```

Dès qu'un objet est instancié, il devient possible de lui appliquer les méthodes de sa classe.

11.7.2 Constructeurs et destructeurs

Une méthode particulière, le *constructeur*, peut être utilisée pour initialiser les attributs d'un objet. Cette méthode doit porter soit le nom que la classe, soit le nom réservé *__construct*. Voici la méthode `Geom()` que l'on pourrait définir pour initialiser les objets de la classe `Geom`.

```
function Geom ($X, $Y, $pNbPoints)
{
 $this->nbPoints = $pNbPoints;
 for ($i = 0; i < $pNbPoints; $i++)
 {
   $this->abcisses[$i] = $X[$i];
   $this->ordonnees[$i] = $Y[$i];
 }
}
```

Le constructeur prend en entrée un tableau d'abcisses, un tableau d'ordonnées, et le nombre de points. On peut alors appeler le constructeur avec `new`.

```
$icone = new Geom (array (12, 32, 56), array (43, 30, 6), 3);
// Calcul de la surface
echo "Surface = " . $icone->surface();
```

On peut également définir un destructeur nommé *__destruct*. Il est appelé quand la dernière variable stockant une référence à un objet est détruite avec `unset()` (ou quand le script est terminé).

Quand on crée une sous-classe, le constructeur et le destructeur de la classe parente ne sont pas appelés automatiquement. C'est au programmeur de le faire, s'il le souhaite, avec la syntaxe `parent::__construct()` et `parent::__destruct()`.

11.7.3 Sous-classes

On peut créer des sous-classes d'une classe. Les objets d'une sous-classe *sont* des objets de la classe parente, mais avec un comportement (ensemble d'attributs et de méthodes) plus détaillé et plus précis. Une sous-classe se définit avec le mot-clé `extends`. On peut par exemple définir des sous-classes `Rectangle`, `Polygone`, etc, pour la classe `Geom`.

```
class Rectangle extends Geom
{
 // Les attributs sont hérités.
 function Rectangle ($X, $Y)
 {
  if (count($X) != 4 or count($Y) != 4)
  {
   echo "Un rectangle a quatre sommets !";
   exit;
  }
  $this->Geom ($X, $Y, 4);
 }

 public function surface ()
 {
  return ($this->abcisses[1] - $this->abcisses[0]) *
       ($this->ordonnees[1] - $this->ordonnees[0]);
 }
}
```

Comme un rectangle est un objet de la classe `Geom`, il *hérite* de tous les attributs de cette classe (soit le tableau des abcisses et des ordonnées) et de toutes ses méthodes. Cependant un rectangle a une définition plus précise qu'un objet géométrique quelconque : il a quatre sommets, à angle droits. Le constructeur de la classe `Rectangle` doit tester que ces contraintes sont vérifiées (réalisé partiellement dans l'exemple ci-dessus) avant d'appeler le constructeur de `Geom` qui initialise les tableaux.

De même, la méthode de calcul d'une surface est beaucoup plus facile à implanter pour un rectangle. On a donc *remplacé* la définition générale de la méthode `surface()` par une implantation plus spécifique que celle de la classe `Geom`. On parle de *surcharge* dans le jargon orienté-objet.

11.7.4 Manipulation des objets

L'opérateur `new` renvoie une *référence* vers un objet instance de la classe. Toutes les manipulations d'un objet en PHP 5 s'effectuent ensuite par l'intermédiaire de

références à cet objet. Il faut imaginer qu'un objet est constitué d'un espace réservé, désigné par une ou plusieurs variables. En particulier :

1. une affectation `$o2 = $o1;` fait de la variable `$o2` une nouvelle référence vers l'objet déjà désigné par `$o1` ;
2. un appel de fonction `fonc($o1)` transmet à la fonction une référence vers l'objet `$o1`.

Toute opération appliquée à un objet par l'intermédiaire de l'une de ses références affecte l'unique zone réservée où les propriétés de l'objet sont stockées. Dans les cas ci-dessus, si on modifie une propriété de l'objet désigné par `$o1`, cette modification est visible par `$o2` puisque les deux variables désignent le même objet. Si la fonction modifie le contenu de l'objet qui lui est passé en paramètre, ce changement reste effectif après la sortie de la fonction puisque la zone stockant l'objet est partagée par la fonction et le script principal.

SPour effectuer une *copie* d'un objet afin d'éviter ces effets parfois non souhaités, il faut utiliser le « clonage » avec le mot-clé `clone` :

```
// Copie de o1 dans o2
$o2 = clone $o1;
```

La copie duplique simplement la zone mémoire représentant `$o1` vers la zone représentant `$o2`. Ce n'est pas toujours la bonne technique car si `$o1` contient lui-même, parmi ses propriétés, des références à d'autres objets, c'est la référence qui va être copiée et pas les objets eux-mêmes. Il est possible de contrôler le comportement de la copie en définissant une méthode `__clone()`.

Deux objets sont *égaux* si les valeurs de leurs propriétés sont les mêmes. Ils sont *identiques* s'ils ont la même identité (autrement dit si les deux variables référencent le même objet). L'opérateur d'égalité est « == » et l'opérateur d'identité est « === ».

11.7.5 Compléments

Plusieurs chapitres de ce livre donnent des exemples concrets de programmation objet : voir les classes `BD`, `Tableau` et `Formulaire` du chapitre 3, les classes et sous-classes de traitement d'un document XML page 339, et les classes du pattern MVC dans le chapitre 6.1. Ces classes illustrent non seulement les concepts orientés-objet et les constructions syntaxiques précédentes, mais elles montrent aussi comment les mettre en œuvre en pratique. Décortiquer une classe existante est souvent le meilleur moyen de comprendre l'approche objet et d'acquérir de bonnes pratiques.

QUATRIÈME PARTIE

Annexes

A

Installation Apache/PHP/MySQL

Les instructions qui suivent permettent de compléter l'installation de MySQL. Elles sont valables aussi bien pour une installation Windows que pour une installation Linux. Les exemples donnés ci-dessous s'appliquent à ce dernier système mais sont aisément transposables à Windows.

A.1 MOT DE PASSE ROOT

Au moment de l'initialisation d'un serveur MySQL, il n'y a que deux utilisateurs : `root@localhost` avec tous les droits sur toutes les bases, et l'utilisateur anonyme `"@localhost` qui n'a aucun droit sur aucune base (sauf pour Windows). Vous pouvez consulter le contenu de la table `user` (voir page 448 pour une description des droits d'accès) avec les commandes suivantes.

```
% mysql -u root

mysql> USE mysql;
Database changed
mysql> SELECT * FROM user;
```

Au départ, il est possible de se connecter sans entrer de mot de passe pour `root`, ce qui est très dangereux. La première chose à faire est d'attribuer un mot de passe à `root` avec la commande :

```
mysql> set password for root@localhost = Password('motdepasse');
```

Bien entendu vous devez choisir un mot de passe raisonnablement difficile à deviner. Vous pouvez alors vous connecter avec la commande suivante :

```
% mysql -u root -p
```

L'utilitaire *mysql* demande le mot de passe à chaque connexion. Il est possible d'éviter d'avoir à effectuer ce contrôle du mot de passe répétitivement en créant un fichier de configuration, décrit dans la section suivante.

L'utilisateur anonyme est une source de confusion, et un problème pour la sécurité sous Windows. Il est recommandé de le supprimer en détruisant cet utilisateur dans la table *user*. Voici les commandes.

```
% mysql -u root -p
Enter password:

mysql> DELETE FROM user WHERE  user='';
```

A.2 CRÉATION DE BASES ET D'UTILISATEURS

Étant connecté sous `root` (attention : il s'agit bien du `root` de MySQL, différent du compte UNIX), on peut créer des bases et des utilisateurs. Un utilisateur MySQL est identifié par deux composants : le *nom* et *l'hôte*. Cela complique assez nettement le système d'accès, puisque deux utilisateurs de même nom ne seront pas traités de la même manière par MySQL selon l'hôte qu'ils utilisent. De plus, on peut utiliser le caractère '%' pour désigner *tous* les hôtes et la chaîne vide '' pour désigner *tous* les utilisateurs. Par exemple :

- `fogg@cartier.cnam.fr` est l'utilisateur de nom `fogg` accédant à MySQL depuis `@cartier.cnam.fr` ;
- `fogg@%.cnam.fr` est l'utilisateur de nom `fogg` accédant à MySQL depuis n'importe quel hôte dont le domaine est `cnam.fr` ;
- `"@magellan.cnam.fr` représente tout utilisateur se connectant à partir de `magellan.cnam.fr` ;

Les utilisateurs sont stockés dans la table *user* qui, en plus des attributs définissant les droits, contient les attributs `Host`, `User` et `Password`. Voici les valeurs de ces attributs pour les utilisateurs de la liste précédente. La table contient également une ligne automatiquement créée par MySQL au moment de l'installation, correspondant à l'utilisateur « anonyme » ayant un droit de connexion à partir de l'ordinateur local.

```
+------------------+------------+------------------+
| host             | user       | password         |
+------------------+------------+------------------+
| magellan.cnam.fr |            | 7c783a0e25967167 |
| %.cnam.fr        | fogg       | 7c78343c25967b95 |
| localhost        | fogg       | 7c786c222596437b |
| localhost        |            |                  |
+------------------+------------+------------------+
```

Le nom `localhost` est un synonyme pour l'ordinateur hôte du serveur *mysqld*, soit en l'occurrence *cartier.cnam.fr*.

La création d'un compte se fait soit en insérant directement dans une des tables de *mysql* avec la commande `INSERT`, soit avec une commande spécialisée `GRANT`. La seconde méthode est préférable car elle est plus claire et, surtout, elle prend effet immédiatement. En effet, quand on utilise un `INSERT` dans une table de *mysql*, la mise à jour est différée et il faut utiliser un ordre `FLUSH PRIVILEGES` pour que la création du compte prenne effet. Nous utilisons systématiquement les commandes `GRANT` et `REVOKE`.

A.2.1 La commande `GRANT`

Voici quelques commandes de création.

```
mysql -u root -p mysql
Enter password:

mysql> GRANT ALL PRIVILEGES ON Films.* to fogg@localhost
    -> IDENTIFIED BY 'mdp1';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT select ON Films.* to fogg@%.cnam.fr
    ->              IDENTIFIED BY 'mdp2';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT select  ON Films.Film to ''@magellan.cnam.fr
    ->              IDENTIFIED BY 'mdp3' ;
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT USAGE ON *.* to visiteur@localhost ;
Query OK, 0 rows affected (0.00 sec)
```

La commande `GRANT` a deux effets complémentaires. D'une part elle crée un utilisateur s'il n'existe pas, d'autre part elle lui accorde des droits sur une ou plusieurs bases, et change éventuellement son mot de passe.

- L'utilisateur `fogg@localhost` se voit donc accorder tous les droits (hors droits d'administration) sur la base *Films*. L'expression `ALL PRIVILEGES` permet d'éviter l'énumération `select`, `insert`, `create`, etc. De même, `Films.*` désigne toutes les tables de la base *Films*, existantes ou à venir.
- L'utilisateur de nom `fogg`, se connectant à MySQL à partir d'un ordinateur situé dans le domaine *cnam.fr*, obtient seulement des droits de lecture sur toutes les tables de la base *Films*.
- Enfin tout utilisateur se connectant (avec un mot de passe `mdp3`) à MySQL à partir de *magellan.cnam.fr* obtient seulement des droits de sélection sur la table *Film*.
- La dernière commande `GRANT` crée un utilisateur `visiteur@localhost` qui n'a aucun droit sur aucune base, et peut se connecter sans mot de passe.

Le mot-clé `USAGE` correspond à un droit de connexion. Cela n'empêche pas d'appeler des fonctions SQL (sans la clause `FROM`), comme le montre l'exemple ci-dessous. En revanche, toute tentative d'accéder à une base avec la commande `USE` est refusée.

```
% mysql -u visiteur

mysql> SELECT USER();
+--------------------+
| USER()             |
+--------------------+
| visiteur@localhost |
+--------------------+
mysql> USE Films;
ERROR 1044: Access denied for user: 'visiteur@localhost'
                    to database 'Films'
```

Il est toujours possible, par la suite, d'accorder des droits supplémentaires ou d'affecter un mot de passe à `visiteur@localhost`.

Utilisateurs du site Films

Dans le cas d'un site web PHP/MySQL, l'utilisateur est explicitement indiqué dans la fonction de connexion PHP `mysql_connect()`, et l'ordinateur à partir duquel la connexion s'effectue est celui qui héberge le serveur Apache. Dans notre cas, *mysqld* et Apache sont sur le même ordinateur. On peut donc se limiter à la création d'utilisateurs MySQL associés à l'hôte `localhost`.

Nous avons besoin de deux comptes utilisateurs pour la base *Films*. Le premier est un compte administrateur de la base, identique à `fogg@localhost`. Il permet de faire des mises à jour, des sélections sur toutes les tables, etc.

Le deuxième utilisateur, `visiteurFilms@localhost` a le droit d'exécuter des requêtes `SELECT` et d'en afficher le résultat. On ne veut pas accorder de droits de mise à jour sur la base *Films* à cet utilisateur. On veut même lui dissimuler une partie de la base qui comporte des informations sensibles ou confidentielles : la table *SessionWeb*, la table *Internaute*, etc.

Le système de droits d'accès de MySQL est assez fin pour permettre d'indiquer très précisément les droits d'accès de `visiteurFilms@localhost`. Voici les commandes `GRANT` contenues dans le fichier *Droits.sql*, fourni avec notre site.

Exemple A.1 *webscope/installation/Droits.sql* : *Script de création des utilisateurs de la base Films*

```
#
# Création des utilisateurs pour la base Films de MySQL
# Ce script doit être exécuté sous un compte administrateur
#

GRANT ALL PRIVILEGES ON Films.* TO adminFilms@localhost
            IDENTIFIED BY 'mdpAdmin';
```

```
GRANT USAGE ON Films.Film
    TO visiteurFilms@'%' IDENTIFIED BY 'mdpVisiteur';
GRANT select ON Films.Film TO visiteurFilms@'%';
GRANT select ON Films.Artiste TO visiteurFilms@'%';
GRANT select ON Films.Role TO visiteurFilms@'%';
GRANT select ON Films.Pays TO visiteurFilms@'%';
GRANT select ON Films.Message TO visiteurFilms@'%';
GRANT select (titre, note) ON Films.Notation TO visiteurFilms@'%';
```

La dernière commande `GRANT` donne des droits de sélection restreints à un sous-ensemble des attributs d'une table. On permet donc l'interrogation de `titre` et `note` dans *Notation*, mais pas de l'attribut `email` que l'on peut considérer comme confidentiel. De même, la table *Internaute*, qui contient des informations personnelles sur les internautes, est inaccessible pour une requête SQL effectuée à partir du site *Films*.

A.2.2 Modification des droits d'accès

La commande inverse de `GRANT ... TO` est `REVOKE ... FROM`. La définition des droits est identique pour `REVOKE`. Voici par exemple comment on retire à l'utilisateur `adminServeur@localhost` le droit de créer ou détruire des tables

```
mysql> REVOKE create, drop ON *.* FROM adminServeur@localhost;
Query OK, 0 rows affected (0.00 sec)
```

Pour affecter un mot de passe, il ne faut surtout pas utiliser la commande `UPDATE`. Les mots de passe sont cryptés avec la fonction `PASSWORD()`. On peut utiliser une instruction spécifique.

```
mysql> SET PASSWORD FOR visiteur@localhost = PASSWORD('monMot');
Query OK, 0 rows affected (0.00 sec)
```

Mais le mieux est de s'en tenir à la commande `GRANT`, avec l'option `USAGE` si on ne veut pas modifier les droits existants pour l'utilisateur.

```
mysql>  GRANT USAGE ON *.* TO visiteur@localhost
              IDENTIFIED BY 'monMot2';
Query OK, 0 rows affected (0.00 sec)
```

En règle générale, pour ajouter des droits à un utilisateur existant, il suffit d'utiliser à nouveau la commande `GRANT`.

A.3 FICHIERS DE CONFIGURATION

Tous les programmes d'accès à MySQL prennent en compte un ou plusieurs fichiers de configuration. Chaque fichier indique des options pour le serveur ou les clients MySQL.

A.3.1 Format d'un fichier de configuration

Un fichier de configuration est au format ASCII, et comprend un ensemble de sections. Chaque section commence par une option [*section*] où *section* peut être `server`, `client`, ou le nom de l'un des programmes clients de MySQL. On trouve ensuite les valeurs des paramètres pour le ou les clients de la section courante.

Un exemple de fichier de configuration, nommé *my-example.cnf*, est fourni dans le répertoire *support-files*. Voici une partie de ce fichier, illustrant la structure des paramètres de configuration (N.B. : le caractère '#' indique une ligne de commentaires).

```
# Pour tous les clients
[client]
port            = 3306
socket          = /tmp/mysql.sock

# Pour le serveur
[mysqld]
port            = 3306
socket          = /tmp/mysql.sock
skip-locking
set-variable    = key_buffer=16M
set-variable    = max_allowed_packet=1M
set-variable    = thread_stack=128K
# Start logging
log

# Pour le client mysql
[mysql]
no-auto-rehash
```

La liste des paramètres acceptés par un programme est donnée par l'option `help`.

A.3.2 Les différents fichiers

Il y a trois fichiers pris en compte. Sous Unix, MySQL accède à ces fichiers dans l'ordre indiqué ci-dessous, et les paramètres trouvés en dernier sont considérés comme prioritaires. Il est possible également de donner, au lancement de chaque programme, une liste de paramètres sur la ligne de commande.

/etc/my.cnf Ce fichier donne les options globales pour *tous* les serveurs MySQL tournant sur la machine. On peut y indiquer des choix sur la taille des buffers utilisés, ou le paramétrage par défaut de tel ou tel client MySQL, mais surtout pas des mots de passe ! Ce fichier est accessible en lecture par tous les clients MySQL.

DATADIR/my.cnf Quand il y a plusieurs serveurs, ce fichier, placé dans le répertoire racine des bases d'un serveur, permet d'indiquer les options pour ce serveur en particulier.

.my.cnf Placé dans le répertoire `$HOME` d'un utilisateur, ce fichier donne les options et préférences de cet utilisateur. On peut y placer le compte de connexion à MySQL (login et mot de passe) en s'assurant que ce fichier n'est lisible que par l'utilisateur.

Sous Windows, le fichier de configuration doit être placé dans *C:*. Malheureusement tout utilisateur ayant un compte sur la machine pourra lire son contenu. Évitez donc d'y placer des mots de passe.

A.3.3 Configuration du serveur

Le fichier */etc/my.cnf* est particulièrement utile pour paramétrer le serveur. Entre autres options importantes, on peut :

1. choisir la langue pour les messages d'erreur ;
2. choisir la taille de la mémoire centrale allouée au serveur ;
3. fixer l'une des très nombreuses options de lancement du serveur *mysqld*.

La commande suivante donne toutes les options possibles,

```
mysqld --help
```

Voici un extrait de l'affichage obtenu avec cette commande.

```
  -b, --basedir=path    Path to installation directory. All paths are
                        usually resolved relative to this
  -h, --datadir=path    Path to the database root
  -L, --language=...    Client error messages in given language. May be
                        given as a full path
  -l, --log[=file]      Log connections and queries to file
  --log-update[=file]   Log updates to file.# where # is a unique number
                        if not given.
  --pid-file=path       Pid file used by safe_mysqld
  -P, --port=...        Port number to use for connection
  -O, --set-variable var=option
                        Give a variable an value. --help lists variables
  --socket=...          Socket file to use for connection
  -u, --user=user_name  Run mysqld daemon as user
```

La version longue des paramètres ci-dessus peut être utilisée dans le fichier */etc/my.cnf*, à l'intérieur de la section commançant par `mysqld`. Voici l'exemple

d'un paramétrage :

```
[mysqld]
port            = 3306
socket          = /tmp/mysql.sock
user            = mysql
set-variable    = key_buffer=64M
language = french
log
log-update
```

On a indiqué, entre autres :

1. que les messages doivent être en français (option `language`) ;
2. que toutes les connexions et requêtes à MySQL sont conservées dans un fichier *log* (dont le nom est, par défaut, *mysqld.log* et l'emplacement le répertoire racine des bases de données) ;
3. que toutes les mises à jour sont également conservées dans un fichier de *log* ;
4. que la mémoire *cache* pour les index est de 64 mégaoctets.

A.3.4 Configuration d'un compte administrateur

Voici comment configurer un compte administrateur qui veut se connecter, avec tout client de MySQL, à la base *mysql* sous le compte `root`. Prenons comme exemple le client *mysql* : voici un extrait des paramètres obtenus par `mysql -help`.

```
-D, --database=..     Database to use
-e, --execute=...     Execute command and quit.
-f, --force           Continue even if we get an sql error.
-H, --html            Produce HTML output
-O, --set-variable var=option
                      Give a variable an value. --help lists variables
-p, --password=...    Password to use when connecting to server
                      If password is not given it's asked from the tty.
-u, --user=#          User for login if not current user
-E, --vertical        Print the output of a query (rows) vertically
```

Dans un environnement Unix, voici comment indiquer avec un fichier de configuration, que l'utilisateur `mysql` est l'administrateur de la base de données. On indique que son compte de connexion à MySQL est `root`, on donne le mot de passe (mettez le vôtre !), et la base par défaut, `mysql`.

1. Copiez *my-example.cnf* dans le répertoire `$HOME` de `mysql`, et renommez-le en *.my.cnf*.
2. Éditez *.my.cnf*. Le fichier contient plusieurs sections dédiées respectivement aux paramètres du serveur, du client, etc. Dans la section commençant par `[client]`, ajoutez deux lignes comme indiqué ci-dessous :

```
[client]
user            = root
password        = mdproot
```

Dans la section `[mysql]`, définissez la base par défaut :

```
[mysql]
database        = mysql
```

3. Il ne reste plus qu'à protéger ce fichier des regards indiscrets

```
% chmod go-rwx .my.cnf
```

Une fois ce fichier créé, la commande `mysql`, exécutée sous le compte UNIX `mysql`, vous connecte directement à la base *mysql* sans avoir à entrer le compte utilisateur, le mot de passe, et sans avoir à utiliser la commande `USE mysql`. Le principe est généralisable à tous les utilisateurs, en leur permettant une connexion automatique à leur base de référence. Vous pouvez par exemple indiquer des options de connexion pour le compte utilisateur d'Apache.

Le compte et le mot de passe sont valables pour tous les programmes clients de MySQL, ce qui permet, sous le compte `mysql`, d'arrêter le serveur sans saisir de mot de passe avec la commande.

```
% mysqladmin shutdown
```

A.4 SAUVEGARDES

Il existe de nombreuses manières de perdre des données. Bien entendu, on pense toujours à un incident matériel comme la panne d'un disque, mais le problème vient beaucoup plus fréquemment d'une erreur humaine. Par exemple il est très facile d'introduire un point-virgule mal placé dans une requête, avec un résultat qui peut être désastreux :

```
% mysql

mysql> DELETE FROM Artiste; WHERE id = 67;
Query OK, 0 rows affected (0.02 sec)

ERROR 1064: You have an error in your SQL syntax
              near 'WHERE id = 0' at line 1
mysql> select * from Artiste;
Empty set (0.00 sec)
```

On a donc bel et bien exécuté la requête `DELETE FROM Artiste`, sans le `WHERE` qui est placé après le point-virgule « ; ». Résultat : tout est détruit dans *Artiste*, sans possibilité de récupération autre qu'une sauvegarde (sauf si on utilise l'option transactionnelle de MySQL, InnoDB, non décrite ici).

> **REMARQUE –** Quand MySQL exécute `DELETE FROM` *`table`*, sans clause `WHERE`, il ne se donne même pas la peine de parcourir les lignes de la table : le fichier est détruit et recréé. D'où le message *Query OK, 0 rows affected,* bien que toutes les lignes aient effectivement disparu.

Les sauvegardes peuvent se faire se manière traditionnelle, en créant une archive *zip* ou *tar* (n'oubliez pas d'arrêter le serveur *mysqld* auparavant) contenant les fichiers de la base. L'inconvénient est qu'il n'est pas facile d'en extraire, si besoin est, une partie seulement des tables ou des lignes.

L'utilitaire phpMyAdmin propose une fonction d'exportation de base de données simple à utiliser, et comprenant toutes les options nécessaires. MySQL seul fournit deux solutions complémentaires, un utilitaire, *mysqldump*, et la création de fichiers *log* qui enregistrent au fur et à mesure les modifications sur la base. L'utilitaire *mysqldump* produit un fichier contenant les ordres SQL de création des tables et/ou des lignes dont la sauvegarde a été demandée. La syntaxe générale est :

```
% mysqldump [options] base [tables]
```

Pour sauvegarder la base *FilmSQL* dans un fichier *filmSQL.sv*, la commande est donc :

```
% mysqldump -u root -p  FilmSQL > filmSQL.sv
```

Bien entendu on peut se connecter sous n'importe quel nom d'utilisateur ayant au moins un droit `select` sur la base. Comme tous les utilitaires de MySQL, *mysqldump* utilise les informations de connexion du fichier de configuration si elles existent.

Pour sauvegarder une ou plusieurs tables, on donne leur nom. Par exemple on peut demander une sauvegarde de la table *Film*.

```
% mysqldump -u root -p  FilmSQL Film > filmSQL.sv
```

Le fichier *filmSQL.sv* contient alors à la fois les commandes de création de la table et les commandes d'insertion des lignes.

```
#
# Table structure for table 'Film'
#
CREATE TABLE Film (
  titre varchar(50) DEFAULT '' NOT NULL,
  annee int DEFAULT '0' NOT NULL,
  id_realisateur int,
  genre varchar(20),
  PRIMARY KEY (titre)
#
# Dumping data for table 'Film'
#

INSERT INTO Film VALUES ('Impitoyable',1992,20,'Western');
INSERT INTO Film VALUES ('Van Gogh',1990,29,'Drame');
INSERT INTO Film VALUES ('Kagemusha',1980,68,'Drame');
INSERT INTO Film VALUES ('Les pleins pouvoirs',1997,20,'Policier');
```

Un des grands avantages de cette méthode est que l'on peut utiliser le fichier comme un script SQL, soit pour recréer une base ou une table dans MySQL, soit

pour transférer une base vers un autre système relationnel. Comme le fichier est au format texte, on peut également facilement en extraire des parties pour récupérer partiellement des données ou des tables.

Les principales options de *mysqldump* sont données dans la table 1.1. Comme pour tous les autres utilitaires, on obtient la liste des options disponibles en lançant :

```
% mysqldump -help
```

Tableau 1.1 — Options de *mysqldump*.

Option	Description
-t	Sauvegarde des lignes, mais pas des commandes de création de table.
-T répertoire	Écrit dans *répertoire* deux fichiers pour chaque table. Le fichier *table.sql* contient la commande de création, et le fichier *table.txt* les lignes.
-c	Produit des ordres INSERT complets, avec la liste des attributs.
-l	Verrouille les tables avant la sauvegarde pour éviter des mises à jour simultanées.
-u, -p, -h	Les options habituelles pour, respectivement, l'utilisateur, le mot de passe et le nom de l'hôte de *mysqld*.

Les options par défaut peuvent changer avec les versions de MySQL. À titre d'illustration, voici la commande utilisée pour sauvegarder la base *Films* dans un fichier *SvFilms* :

```
mysqldump -u adminFilms -pmdpAdmin -t Films Artiste Internaute\
             Film Notation Role SequenceArtiste \
            --skip-opt -c --skip-lock-tables \
           --default-character-set=latin1 > SvFilms
```

Le fichier obtenu est compatible avec PostgreSQL et SQLite. L'utilitaire propose des exports aux formats acceptés par d'autres SGBD, dont ORACLE.

A.5 phpMyAdmin

Il existe de nombreux outils qui facilitent (au moins pour une prise de contact) la maintenance d'une installation MySQL. Le plus populaire est phpMyAdmin, une interface d'administration écrite en PHP.

phpMyAdmin est une très bonne illustration de l'utilisation de PHP en association avec MySQL, et peut s'utiliser aussi bien sous Linux que sous Windows. Il se présente sous la forme d'un ensemble de fichiers PHP. Le fichier `Documentation.html` propose une documentation assez brève.

phpMyAdmin est configurable avec le fichier *config.inc.php*. La première chose à faire est d'affecter l'URL de *phpMyAdmin* à l'élément `PmaAbsoluteUri` du tableau `$cfg`. Par exemple :

```
$cfg['PmaAbsoluteUri'] = 'http://localhost/phpMyAdmin/';
```

Le tableau `$cfg` permet de configurer *phpMyAdmin*, notamment pour la partie `$cfg['Servers']['1']` qui donne les paramètres de connexion à MySQL. Voici les plus importants, avec leur valeur par défaut :

```
// Serveur MySQL
$cfg['Servers'][$i]['host']          = 'localhost';
// Port MySQL
$cfg['Servers'][$i]['port']          = '';
// Socket MySQL
$cfg['Servers'][$i]['socket']        = '';
// Mode de connexion à MySQL
$cfg['Servers'][$i]['connect_type']  = 'tcp';
// Utilisateur contrôlant les accès
$cfg['Servers'][$i]['controluser']   = '';
// Mot de passe utilisateur contrôleur
$cfg['Servers'][$i]['controlpass']   = '';
// Mode d'authentification
$cfg['Servers'][$i]['auth_type']     = 'config';
// Utilisateur MySQL
$cfg['Servers'][$i]['user']          = 'root';
// Mot de passe utilisateur
$cfg['Servers'][$i]['password']      = '';
// Si indiqué, donne la seule base accessible
$cfg['Servers'][$i]['only_db']       = '';
```

Les options `host`, `port` et `socket` permettent de préciser l'hôte du serveur MySQL, le port d'écoute et le nom de la *socket* d'accès à MySQL. En principe, les valeurs par défaut conviennent. L'option `auth_type` détermine le type de protection utilisé pour l'accès à phpMyAdmin.

- **Cas `auth_type='config'`**
  Dans ce cas la connexion à MySQL se fait avec les valeurs des champs `user` et `password`. Il faut renseigner ces champs avec un compte utilisateur. Par défaut, l'utilisateur `root` (utilisateur MySQL) sans mot de passe est indiqué, ce qu'il faut impérativement changer pour un site en production.
- **Cas où `auth_type` vaut `'http'` ou `'cookie'`**
  Dans ce cas *phpMyAdmin* transmet au programme client, au moment de la première demande d'accès d'un utilisateur, un document avec un en-tête HTML indiquant que l'utilisateur doit s'identifier. Le navigateur produit alors une fenêtre demandant un nom et un mot de passe, transmis à *phpMyAdmin* quand l'utilisateur les a saisis.
  phpMyAdmin vérifie que ce compte correspond à un compte MySQL valide. Si c'est le cas, une session est ouverte. Cette session est gérée soit avec les variables standard d'authentification de HTTP (mode `'http'`), soit par des *cookies*.

Pour vérifier qu'un compte d'accès est correct, *phpMyAdmin* doit disposer d'un autre compte pouvant interroger la base dictionnaire `mysql`. Ce deuxième compte doit être indiqué dans les champs `controluser` et `controlpass`. On peut donner

le compte `root` qui a tous les droits sur la base `mysql`, mais il est gênant de mettre en clair le mot de passe de `root` dans un fichier. Un moindre mal est de créer un compte MySQL spécial *phpAdmin*, qui a seulement le droit d'inspecter les tables de `mysql`.

```
mysql> GRANT select ON mysql.* TO phpAdmin
    ->                IDENTIFIED by'mdpPhpAdmin';
```

Voici maintenant le paramétrage pour l'authentification d'un utilisateur au moment de la connexion.

```
$cfg['Servers'][$i]['controluser']   = 'phpAdmin';
$cfg['Servers'][$i]['controlpass']   = 'mdpPhpAdmin';
$cfg['Servers'][$i]['auth_type']     = 'coookie';
```

phpMyAdmin se connectera avec le compte `phpAdmin/mdpPhpAdmin` pour vérifier le compte saisi interactivement par l'utilisateur.

Comme tout fichier contenant des mots de passe, *config.inc.php* doit être protégé des regards indiscrets. Limitez les droits en lecture au compte utilisateur d'Apache, et les droits en écriture au compte du webmestre du site.

B

Référence MySQL

Cette annexe est consacrée aux commandes, langages et utilitaires de MySQL. Nous commençons par les types de données utilisables pour les attributs des tables, avant de passer en revue les principales commandes du langage SQL et des extensions proposées par MySQL.

Cette annexe ne constitue pas une référence complète. MySQL, dans sa version 5, constitue un SGBD très riche dont la présentation détaillée mérite un ouvrage complet. Nous avons renoncé à entrer dans des débats très pointus et assez éloignés de notre sujet. Ce qui suit constitue donc une sélection des commandes les plus usuelles, et couvre en tout cas toutes les fonctionnalités abordées précédemment dans ce livre, largement suffisantes pour la plupart des sites web.

B.1 TYPES DE DONNÉES MySQL

MySQL est remarquablement conforme à la norme SQL ANSI, contrairement à d'autres SGBD, plus anciens, dont le système de types était déjà bien établi avant la parution de cette norme (en 1992). MySQL propose également quelques variantes et extensions, la principale étant la possibilité de stocker des attributs de type *Binary Long Object* (`BLOB`).

Le tableau 2.1 résume la liste des types d'attributs, donne la taille de stockage utilisée par MySQL, et indique si le type fait partie ou non de la norme SQL ANSI.

Tableau 2.1 — Types de MySQL

Type	Taille en octets	SQL ANSI ?
TINYINT	1	Non
SMALLINT	2	Oui
MEDIUMINT	3	Non
INTEGER	4	Oui
BIGINT	8	Oui
FLOAT	4	Oui
DOUBLE PRECISION	8	Oui
REAL	8	Oui
NUMERIC (*M, D*)	*M*, (*D*+2 si $M < D$)	Oui
DECIMAL (*M, D*)	*M*, (*D*+2 si $M < D$)	Oui
CHAR(*M*)	*M*	Oui
VARCHAR(*M*)	*L+1* avec $L \leqslant M$	Oui
TINYBLOB/TINYTEXT	< 255	Non
BLOB/TEXT	$< 2^{16}$	Non
MEDIUMBLOB/MEDIUMTEXT	$< 2^{24}$	Non
LONGTEXT/LONGBLOB	$< 2^{32}$	Non
DATE	3	Oui
TIME	3	Oui
DATETIME	8	Oui
TIMESTAMP	4	Oui
YEAR	1	Oui
ENUM	au plus 2	Non
SET	au plus 8	Non

Types numériques exacts

La norme SQL ANSI distingue deux catégories d'attributs numériques : les *numériques exacts*, et les *numériques flottants*. Les types de la première catégorie (essentiellement `INTEGER` et `DECIMAL`) permettent de spécifier la précision souhaitée pour un attribut numérique, et donc de représenter une valeur exacte. Les numériques flottants correspondent aux types couramment utilisés en programmation (`FLOAT`, `DOUBLE`) et ne représentent une valeur qu'avec une précision limitée.

Tous les types numériques acceptent l'option `ZEROFILL` qui indique que l'affichage d'une valeur numérique se fait avec la largeur maximale, les chiffres étant complétés par des zéros. Par exemple la valeur 3 stockée dans un type `INTEGER ZEROFILL` sera affichée `0000000003`.

Le type `INTEGER` permet de stocker des entiers sur 4 octets. La taille de l'affichage est fonction de la valeur maximale possible (en l'occurrence, 10 positions), mais peut être précisée optionnellement avec la valeur *M* comme indiqué dans la syntaxe ci-dessous.

```
INTEGER[(M)] [UNSIGNED] [ZEROFILL]
```

Le type `INTEGER` peut être complété par le mot-clé `UNSIGNED` qui spécifie si le stockage comprend un bit de signe ou non. Cette option (qui ne fait pas partie de la norme SQL) a un impact sur l'intervalle des valeurs possibles. Par exemple `INTEGER UNSIGNED` pourra stocker des valeurs dans l'intervalle $[0, 2^{32}-1]$, tandis que `INTEGER` correspond à l'intervalle $[-2^{31}, 2^{31}-1]$.

Il existe de nombreuses variantes du type `INTEGER` : `TINYINT`, `SMALLINT`, `MEDIUMINT`, `BIGINT`, avec la même syntaxe et les mêmes options. Ces types diffèrent par la taille utilisée pour le stockage : voir le tableau 2.1.

Le type `DECIMAL` (*M*, *D*) correspond à un numérique de taille maximale *M*, avec un nombre de décimales fixé à *D*. L'option `ZEROFILL` est acceptée, mais pas l'option `UNSIGNED`. La syntaxe est donc :

```
DECIMAL (M, D) [ZEROFILL]
```

Le type `NUMERIC` est un synonyme pour `DECIMAL`. Ces types sont surtout utiles pour manipuler des valeurs dont la précision est connue, comme les valeurs monétaires. Afin de préserver cette précision, MySQL les stocke comme des chaînes de caractères.

Types numériques flottants

Ces types s'appuient sur la représentation des numériques flottants propre à la machine, en simple ou double précision. Leur utilisation est donc analogue à celle que l'on peut en faire dans un langage de programmation comme le C.

1. Le type `FLOAT` correspond aux flottants en simple précision.
2. Le type `DOUBLE PRECISION` correspond aux flottants en double précision. Le raccourci `DOUBLE` est accepté.
3. Le type `REAL` est un synonyme pour `DOUBLE`.

La syntaxe complète de MySQL pour `FLOAT` (identique pour `DOUBLE` et `REAL`) est :

```
FLOAT[(M, D)] [ZEROFILL]
```

où les options *M* et *D* indiquent respectivement la taille d'affichage et le nombre de décimales.

Caractères et chaînes de caractères

Les deux types principaux de la norme ANSI, disponibles également dans MySQL et la plupart des SGBD relationnels, sont `CHAR` et `VARCHAR`. Dans MySQL, ils permettent de stocker des chaînes de caractères d'une taille maximale fixée par le paramètre *M*, *M* devant être inférieur à 255. Les syntaxes sont identiques. Pour le premier :

```
CHAR(M) [BINARY]
```

et pour le second :

```
VARCHAR(M) [BINARY]
```

La différence essentielle entre les deux types est qu'une valeur `CHAR` a une taille fixée. MySQL la complète avec des blancs si sa taille est inférieure à *M*. En revanche une valeur `VARCHAR` a une taille variable et MySQL la tronque après le dernier caractère non blanc.

Si on n'utilise pas l'option `BINARY`, les comparaisons de chaînes de caractères ne distinguent pas les minuscules des majuscules (donc `'AAA'` sera considéré comme identique à `'aaa'`).

Pour stocker des chaînes de caractères plus longues que 255, il faut utiliser une des variantes des types `BLOB` ou `TEXT`. Les comparaisons sur une valeur de type `TEXT` ne distinguent pas majuscules et minuscules. `TEXT` et `BLOB` peuvent donc être considérés comme des `VARCHAR` de grande taille, avec l'option `BINARY` pour le second. Les variantes `TINY`, `MEDIUM`, `LONG` des types `BLOB` et `TEXT` diffèrent par la taille des valeurs acceptées : voir tableau 2.1.

La norme SQL ne propose ni `BLOB`, ni `TEXT`, mais un type `BIT VARYING` qui correspond également à de longues chaînes de caractères.

Dates

Un attribut de type `DATE` stocke les informations « jour », « mois » et « année » (sur 4 chiffres). La représentation interne à MySQL occupe 3 octets, mais les dates sont affichées par défaut au format `AAAA-MM-JJ`. Les nombreuses opérations de conversion de la fonction `DATE_FORMAT()` permettent d'obtenir un format d'affichage quelconque (voir page 475).

Un attribut de type `TIME` stocke les informations « heure », « minute » et « seconde ». L'affichage se fait par défaut au format `HH:MM:SS`. Le type `DATETIME` permet de combiner une date et un horaire, l'affichage se faisant au format `AAAA-MM-JJ HH:MM:SS`.

Le type `TIMESTAMP` stocke une date et un horaire sous la forme du nombre de secondes écoulées depuis le premier janvier 1970. La syntaxe est `TIMESTAMP[(M)]` où *M* indique optionnellement la longueur de l'affichage (mais pas la taille de la représentation interne). Par défaut, elle est fixée à 14, ce qui permet d'afficher un `TIMESTAMP` au format `AAAAMMJJHHMMSS`.

Le comportement d'un attribut de type `TIMESTAMP` est particulier : si de tels attributs ne sont pas spécifiés explicitement dans un ordre SQL d'insertion (`INSERT`) ou de mise à jour (`UPDATE`), MySQL leur affecte automatiquement comme valeur le moment de la mise à jour. Il s'agit alors véritablement d'un « estampillage » de chaque ligne par le moment de la dernière modification affectant cette ligne.

Enfin, le type `YEAR[2|4]` permet de stocker des années sur 2 ou 4 positions.

Les types ENUM et SET

Ces deux types sont particuliers à MySQL. Le premier permet d'indiquer un type énuméré dont les instances ne peuvent prendre leur (unique) valeur que dans un ensemble explicitement spécifié. La syntaxe est :

```
ENUM ('valeur1', 'valeur2', ... 'valeurN')
```

MySQL contrôle, au moment de l'affectation d'une valeur à un attribut de ce type, qu'elle appartient bien à l'ensemble énuméré {*'valeur1'*, *'valeur2'*, ... *'valeurN'*}. MySQL stocke alors l'indice de la valeur, sur 1 ou 2 octets selon la taille de l'ensemble énuméré (au maximum 65535 valeurs).

Le type SET est déclaré comme le type ENUM, mais un attribut de type SET (*'valeur1'*, *'valeur2'*, ... *'valeurN'*) peut prendre *plusieurs* valeurs dans l'ensemble {*'valeur1'*, *'valeur2'*, ... *'valeurN'*}. Il peut y avoir au maximum 64 valeurs dans l'ensemble, et MySQL stocke un masque de bits sur 8 octets au plus pour représenter la valeur d'un attribut.

B.2 COMMANDES DE MySQL

Cette section présente l'ensemble des commandes de MySQL. Tous les termes en *italiques* indiquent des constructions syntaxiques qui reviennent dans plusieurs commandes et sont détaillées séparément.

Commande ALTER TABLE

```
ALTER [IGNORE] TABLE nomTable commandeAlter

commandeAlter :
   ADD [COLUMN] commandeCréation [FIRST | AFTER nomAttribut]
ou ADD INDEX [nomIndex] (nomAttribut,...)
ou ADD PRIMARY KEY (nomAttribut,...)
ou ADD UNIQUE [nomIndex] (nomAttribut,...)
ou ALTER [COLUMN] nomAttribut {SET DEFAULT literal | DROP DEFAULT}
ou CHANGE [COLUMN] ancienNomColonne commandeCréation
ou MODIFY [COLUMN] commandeCréation
ou DROP [COLUMN] nomAttribut
ou DROP PRIMARY KEY
ou DROP INDEX nomIndex
ou RENAME [AS] nomTable
ou optionTable
```

MySQL commence par copier la table existante, effectue les modifications demandées sur la copie, et remplace l'ancienne table par la nouvelle si tout s'est bien passé.

L'option `IGNORE` indique à MySQL de ne pas annuler la commande si celle-ci crée des doublons de clés. Si `IGNORE` est employée, seule la première ligne de chaque ensemble de doublons est prise en compte. Les commandes de modification de schéma ont été illustrées page 204 et suivantes.

Commande `CREATE DATABASE`

```
CREATE DATABASE nomBase
```

MySQL crée un répertoire vide de nom *nomBase*.

Commande `DROP DATABASE`

```
DROP DATABASE [IF EXISTS] nomBase
```

Cette commande détruit *tous* les fichiers d'une base ! Le script s'interrompt si la base n'existe pas, sauf si l'option `IF EXISTS` est utilisée.

Commande `CREATE TABLE`

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] nomTable (commandeCréation,...)
   [optionsTable] [optionSelect]

commandeCréation :
   nomAttribut type [NOT NULL | NULL]
       [DEFAULT literal] [AUTO_INCREMENT]
       [PRIMARY KEY] [référence]
ou PRIMARY KEY (nomAttribut,...)
ou KEY [nomIndex] (nomAttribut,...)
ou INDEX [nomIndex] (nomAttribut,...)
ou UNIQUE [INDEX] [nomIndex] (nomAttribut,...)
ou [CONSTRAINT contrainte]
       FOREIGN KEY nomIndex (nomAttribut,...) [référence]
ou CHECK (expression)

optionsTable :
   TYPE = {ISAM | MYISAM | HEAP | InnoDB}
ou AUTO_INCREMENT = #
ou AVG_ROW_LENGTH = #
ou CHECKSUM = {0 | 1}
ou COMMENT = "commentaires"
ou MAX_ROWS = #
ou MIN_ROWS = #
ou PACK_KEYS = {0 | 1}
ou PASSWORD = "motDePasse"
ou DELAY_KEY_WRITE = {0 | 1}
```

```
optionSelect :
   [IGNORE | REPLACE] SELECT ... (requête SQL)

référence :
   REFERENCES nomTable [(nomAttribut,...)]
       [MATCH FULL | MATCH PARTIAL]
       [ON DELETE optionRéf]
       [ON UPDATE optionRéf]

optionRéf :
   RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```

Cette commande crée une table dans la base courante : voir page 197. L'option `TEMPORARY` indique que la table est créée pour la connexion courante uniquement. L'option *optionTable* permet de choisir le type de table. Le type `InnoDB` est le stockage le plus sophistiqué : il supporte les transactions et les clés étrangères.

Le nom d'un index est optionnel. En son absence, MySQL engendre automatiquement un nom constitué du nom du premier attribut suivi éventuellement des suffixes '_1', '_2', etc. La commande `SHOW INDEX FROM` *nomTable* donne la liste des index sur une table.

Il est possible de créer un index sur une partie seulement d'un attribut avec la syntaxe *nomAttribut(taille)*. C'est obligatoire pour les types `BLOB` et `TEXT`.

Les options `FOREIGN KEY`, `CHECK` et `CONSTRAINT` existent pour des raisons de compatibilité avec SQL ANSI, mais sont ignorées par MySQL (voir chapitre 4).

L'option *optionSelect* permet d'ajouter à la table les attributs d'un ordre SQL.

Commande `OPTIMIZE TABLE`

```
OPTIMIZE TABLE nomTable
```

Cette commande réorganise une table en récupérant les espaces vides et en défragmentant la table. On peut aussi utiliser l'utilitaire *myisamchk*.

Commande `DELETE`

```
DELETE [LOW_PRIORITY] FROM nomTable
   [WHERE clauseWhere] [LIMIT nbLignes]
```

Cette commande détruit toutes les lignes vérifiant les critères de la clause `WHERE`. L'option `LOW_PRIORITY` indique à MySQL que les destructions sont moins prioritaires que toutes les requêtes courantes qui accèdent à la table.

Commande SELECT

```
SELECT [STRAIGHT_JOIN] [SQL_SMALL_RESULT]
       [SQL_BIG_RESULT] [HIGH_PRIORITY]
       [DISTINCT | DISTINCTROW | ALL]
   listeAttributs
   [INTO {OUTFILE | DUMPFILE} 'nomFichier' optionsExport]
   [FROM clauseFROM
   [WHERE clauseWHERE]
   [GROUP BY nomAttribut,...]
   [HAVING clauseWHERE]
   [ORDER BY {entier | nomAttribut| formule} [ASC | DESC] ,...]
   [LIMIT [début,] nbLignes]
   [PROCEDURE nomProcédure] ]

clauseFROM:
   nomTable, nomTable
ou nomTable [CROSS] JOIN nomTable
ou nomTable INNER JOIN nomTable
ou nomTable STRAIGHT_JOIN nomTable
ou nomTable LEFT [OUTER] JOIN nomTable ON expression
ou nomTable LEFT [OUTER] JOIN nomTable USING (listeAttributs)
ou nomTable NATURAL LEFT [OUTER] JOIN nomTable
ou nomTable LEFT OUTER JOIN nomTable ON expression
```

Cette commande extrait d'une ou plusieurs tables les lignes qui satisfont la clause `WHERE`. *`listeAttributs`* est une liste d'attributs provenant des tables du `FROM`, ou d'expressions impliquant des fonctions. On peut faire référence à un attribut par son nom, par le le nom de sa table et son nom, ou par le nom de sa base, le nom de sa table et son nom : `Films.Acteur.nom` désigne l'attribut `nom` de la table *Acteur* de la base *Films*.

Les options suivantes sont des extensions de MySQL :

- `STRAIGHT_JOIN` indique que la jointure doit accéder aux tables dans l'ordre indiqué.
- `SQL_SMALL_RESULT` prévient MySQL que le résultat contiendra peu de lignes, ce qui permet d'optimiser l'exécution de la requête.
- `SQL_BIG_RESULT` indique l'inverse.
- `HIGH_PRIORITY` demande l'exécution de la requête en priorité par raport à celles qui effectuent des modifications.

Ces options sont réservées aux utilisateurs avertis et doivent être utilisées en connaissance de cause.

L'option `INTO OUTFILE` est complémentaire de `LOAD DATA` et accepte les mêmes options. Avec l'option `DUMPFILE`, MySQL écrira une seule ligne dans le fichier, sans caractères de terminaison de lignes ou de colonnes.

La clause `FROM` consiste habituellement en une liste des tables que l'on veut joindre, séparées par des virgules. Un nom de table dans le `FROM` peut aussi être préfixé par le nom de la base (exemple : `Films.Acteur`). On peut remplacer la virgule par le mot-clé `JOIN` et ses variantes : voir le chapitre 10.

Commande `INSERT`

```
   INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
       [INTO] nomTable [(nomAttribut,...)]
       VALUES (expression,...),(...),...
ou INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
       [INTO] nomTable [(nomAttribut,...)]
       SELECT ...
ou INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
       [INTO] nomTable
       SET nomAttribut=expression, ...
```

`INSERT` insère une ou plusieurs lignes dans une table. MySQL propose trois formes de cette commande. La première et la troisième insèrent des valeurs données explicitement. La seconde (`INSERT-SELECT`) insère dans la table le résultat d'un ordre `SELECT`. Cet ordre ne doit contenir ni clause `ORDER BY`, ni référence à la table dans laquelle on insère.

Les valeurs insérées doivent satisfaire les contraintes de la table, notamment les `NOT NULL`. Les attributs non spécifiés dans `INSERT` prennent leur valeur par défaut, ou `NULL` si c'est possible.

L'option `LOW_PRIORITY` indique à MySQL que les insertions sont moins prioritaires que toutes les requêtes courantes qui accèdent à la table. Cela peut bloquer le script pour un temps arbitrairement long. L'option `DELAYED` place les lignes à insérer en liste d'attente. Le script ou le programme peut alors continuer son exécution sans attendre que les lignes aient effectivement été insérées. Enfin, `IGNORE` indique que les lignes qui engendreraient des doublons dans la table sont ignorées.

Commande `REPLACE`

```
   REPLACE [LOW_PRIORITY | DELAYED] [IGNORE]
       [INTO] nomTable [(nomAttribut,...)]
       VALUES (expression,...),(...),...
ou REPLACE [LOW_PRIORITY | DELAYED] [IGNORE]
       [INTO] nomTable [(nomAttribut,...)]
       SELECT ...
```

```
ou REPLACE [LOW_PRIORITY | DELAYED] [IGNORE]
       [INTO] nomTable
       SET nomAttribut=expression, ...
```

`REPLACE` est identique à `INSERT`, à ceci près que les lignes existantes sont remplacées par les nouvelles, alors que l'existence d'une ligne engendre normalement une erreur avec `INSERT`.

Commande `LOAD DATA`

```
LOAD DATA [LOW_PRIORITY] [LOCAL] INFILE 'nomFichier' [REPLACE | IGNORE]
   INTO TABLE nomTable
   [FIELDS
   [TERMINATED BY caractère]
   [OPTIONALLY] ENCLOSED BY caractère]
   [ESCAPED BY caractère]]
   [LINES TERMINATED BY caractère]
   [IGNORE entier LINES]
   [(nomAttributs, ...)]
```

Cette commande permet d'importer un fichier texte dans une table de MySQL et a été présentée page 29. Elle est complémentaire de `SELECT ... INTO OUTFILE` et peut aussi être utilisée par l'intermédiaire de l'utilitaire *mysqlimport*.

Le fichier doit être organisé en *lignes*, découpées en *champs* (*fields* en anglais).

1. Une ligne est marquée par un caractère qui est, par défaut, le code standard de fin de ligne. On peut spécifier un caractère avec `LINES TERMINATED BY`.
2. Les champs sont, par défaut, séparés par des tabulations. On peut indiquer le séparateur avec `TERMINATED BY`. Une autre manière de marquer les champs est de les encadrer avec un caractère, spécifié par `ENCLOSED BY`. Par défaut il n'y a pas de caractère d'encadrement. Enfin, il faut prendre en compte le fait que les caractères de terminaison ou d'encadrement peuvent apparaître dans les champs eux-mêmes. Afin d'indiquer qu'ils n'ont pas de signification, on les préfixe alors par un caractère d'échappement, qui est par défaut « \ », et peut être spécifié avec `ESCAPED BY`.

En résumé, la commande `LOAD DATA` considère par défaut des fichiers où chaque ligne de la table correspond à une ligne du fichier, les champs étant marqués par des tabulations. Si une valeur contient un caractère de fin de ligne ou une tabulation, ce caractère est précédé de « \ » afin d'indiquer qu'il ne doit pas être interprété.

Si l'on indique `OPTIONALLY` dans l'option `ENCLOSED BY`, MySQL accepte que seuls certains champs soient encadrés. Par exemple on peut vouloir encadrer les chaînes de caractères par « " » pour éviter les ambiguïtés liées aux caractères blancs, et ne pas encadrer les numériques, pour lesquels le problème ne se pose pas.

Les valeurs à `NULL` sont indiquées par \N quand il n'y a pas de caractère d'encadrement, par `NULL` sinon. Attention : \N est différent de la chaîne de caractères `'NULL'` !

Le fichier peut être situé sur la machine du serveur MySQL (choix par défaut), ou sur la machine du client (option `LOCAL`). Dans le premier, cas MySQL recherche par défaut dans le répertoire des fichiers de la base courante. Il est bien entendu possible d'indiquer un chemin d'accès absolu.

L'option `LOW_PRIORITY` est identique à celle utilisée pour des `INSERT`. `IGNORE` et `REPLACE` ont les mêmes significations que pour `INSERT` et `REPLACE`. Enfin, `IGNORE` permet de ne pas prendre en compte les premières lignes du fichier, ce qui peut être utile si ce dernier contient un en-tête par exemple.

Si le nombre ou l'ordre des champs dans les lignes du fichier ne correspond pas au nombre ou à l'ordre des attributs de la table, il faut préciser la correspondance entre les champs du fichier et les attributs en donnant la liste de ces derniers à la fin de la commande.

Commande `UPDATE`

```
UPDATE [LOW_PRIORITY] nomTable
   SET nomAttribut=expression, ...
   [WHERE clauseWHERE] [LIMIT entier]
```

La commande `UPDATE` effectue des mises à jour sur les attributs de toutes les lignes satisfaisant les critères du `WHERE`. L'option `LOW_PRIORITY` indique à MySQL que les mises à jour sont moins prioritaires que toutes les requêtes courantes qui accèdent à la table.

Commande `USE`

```
USE nomBase
```

Indique la base courante. Toutes les tables utilisées dans un ordre SQL seront alors considérées comme appartenant à cette base. Pour accéder à une table en dehors de la base courante, on peut préfixer son nom par le nom de la base (voir la commande `SELECT`).

Commande `FLUSH`

```
FLUSH optionFlush, ...

optionFlush :
   HOSTS | LOGS | PRIVILEGES | TABLES | STATUS
```

Cette commande est réservée aux utilisateurs ayant le privilège `reload`. Elle permet de réinitialiser des parties de la mémoire cache de MySQL. On peut également utiliser l'utilitaire *mysqladmin*.

Commande KILL

```
KILL idThread
```

Chaque connexion à MySQL engendre un nouveau *thread*. La commande `KILL` permet de tuer l'un de ces *thread*. Cette commande est réservée aux utilisateurs ayant le privilège `process`.

Commande SHOW

```
SHOW DATABASES [LIKE expression]
ou SHOW TABLES [FROM nomBase] [LIKE expression]
ou SHOW COLUMNS FROM nomTable [FROM nomBase] [LIKE expression]
ou SHOW INDEX FROM nomTable [FROM nomBase]
ou SHOW STATUS
ou SHOW VARIABLES [LIKE expression]
ou SHOW PROCESSLIST
ou SHOW TABLE STATUS [FROM nomBase] [LIKE expression]
ou SHOW GRANTS FOR utilisateur
```

`SHOW` est la commande qui permet d'obtenir des informations sur le schéma d'une base de données (description des tables, des index, des attributs) et sur les utilisateurs MySQL. L'option `LIKE expression` fonctionne comme la clause `LIKE` de la commande `SELECT`, *`expression`* pouvant utiliser les caractères spéciaux « _ » et « % ».

L'option `STATUS` affiche un ensemble de statistiques concernant l'état du serveur et son historique, tandis que l'option `VARIABLES` montre les paramètres d'initialisation du serveur.

Outre l'utilitaire *mysql*, ces commandes sont pour la plupart accessibles avec *mysqlshow*.

Commande EXPLAIN

```
EXPLAIN nomTable
ou EXPLAIN SELECT ...
```

`EXPLAIN nomTable` est un synonyme de `SHOW COLUMNS` ou `DESCRIBE`. `EXPLAIN SELECT` donne le plan d'exécution d'une requête `SELECT`.

Commande DESCRIBE

```
{DESCRIBE | DESC} nomTable {nomAttribut}
```

Permet d'obtenir la description d'une table ou d'un nom d'attribut dans une table. Ce dernier peut contenir les caractères « % » et « _ » avec la même interprétation que dans la commande LIKE.

Commande LOCK TABLES

```
LOCK TABLES nomTable, [AS alias]
       {READ | [LOW_PRIORITY] WRITE}
       [, nomTable READ | [LOW_PRIORITY] WRITE ...]
```

La commande LOCK permet d'effectuer manuellement les verrouillages, normalement automatiques dans un système transactionnel. On peut placer des verrous en lecture ou en écriture et avec différents niveaux de priorité. Pour un non spécialiste des problèmes de concurrence d'accès, l'utilisation de cette commande est peu recommandée car elle a pour effet de bloquer d'autres utilisateurs

Commande UNLOCK TABLES

```
UNLOCK TABLES
```

Relâche tous les verrous détenus pour la session courante.

Commande SET

```
SET [OPTION] sqlOption = valeur, ...
```

Permet de donner une valeur à l'un des paramètres suivant pour la session courante.

- CHARACTER SET *nomCS* | DEFAULT. Définit le jeu de caractères utilisé pour interpréter les chaînes de caractères.
- PASSWORD [FOR *utilisateur*] = PASSWORD()('motDePasse'). Permet de changer un mot de passe pour un utilisateur. On peut également utiliser la commande GRANT : voir page 448.
- SQL_AUTO_IS_NULL = 0 | 1. Si le paramètre est à 1, on peut obtenir la dernière ligne insérée dans une table avec un attribut AUTO_INCREMENT avec la clause

  WHERE *auto_increment_column* IS NULL

- `SQL_BIG_TABLES = 0 | 1`. Un paramètre booléen qui, s'il est à 1, indique que les tables temporaires doivent être stockées sur disque.
- `SQL_BIG_SELECTS = 0 | 1`. Si le paramètre est à 0, MySQL interrompra les jointures qui manipulent plus de `max_join_size` lignes, ce dernier étant un paramètre du serveur.
- `SQL_LOW_PRIORITY_UPDATES = 0 | 1`. Si le paramètre vaut 1, les mises à jour sur une table attendent qu'il n'y ait plus d'utilisateurs effectuant des lectures.
- `SQL_SELECT_LIMIT = value | DEFAULT`. Permet de fixer une limite au nombre de lignes ramenées par un `SELECT`.
- `SQL_LOG_OFF = 0 | 1`. Si le paramètre vaut 1, un utilisateur avec le droit `process` n'engendre pas de mise à jour du fichier *log* standard.
- `SQL_LOG_UPDATE = 0 | 1`. Si le paramètre vaut 1, un utilisateur avec le droit `process` n'engendre pas de mise à jour du fichier *log* des mises à jour.
- `TIMESTAMP = valeur | DEFAULT`. Modifie la valeur du *timestamp*.
- `LAST_INSERT_ID = valeur`. Définit la valeur retournée par le prochain appel à `LAST_INSERT_ID()`.
- `INSERT_ID = valeur`. Définit la valeur à utiliser lors de la prochaine insertion d'un attribut `AUTO_INCREMENT`.

Commande GRANT

```
GRANT privilège [(listeAttributs)]
      [,privilège [(listeAttributs)] ]
   ON {nomTable| * | *.* | nomBase.*}
   TO nomUtilisateur [IDENTIFIED BY 'motDePasse']
   [, nomUtilisateur [IDENTIFIED BY 'motDePasse'] ...]
   [WITH GRANT OPTION]
```

Définit et modifie les droits d'accès des utilisateurs. Voir page 448.

Commande REVOKE

```
REVOKE privilège [(listeAttributs)]
      [,privilège [(listeAttributs)] ]
   ON {nomTable| * | *.* | nomBase.*}
   FROM nomUtilisateur [, nomUtilisateur ...]
```

Retire des droits à un ou plusieurs utilisateur(s). Voir page 448.

Commande `CREATE INDEX`

```
CREATE [UNIQUE] INDEX nomIndex
       ON nomTable (nomAttribut[(taille)],... )
```

Création d'un index. Voir page 204.

Commande `DROP INDEX`

```
DROP INDEX nomIndex ON nomTable
```

Suppression d'un index.

B.3 FONCTIONS MySQL

Les fonctions suivantes peuvent être utilisées dans des requêtes SQL. Pour la plupart, elles constituent un ajout de MySQL à la norme SQL ANSI.

ABS (*nombre*)

Renvoie la valeur absolue de *nombre*

ACOS (*nombre*)

Renvoie le cosinus inverse de *nombre*, exprimé en radians

ASCII (*car*)

Renvoie le code ASCII du caractère *car*

ASIN (*nombre*)

Renvoie le sinus inverse de *nombre*, exprimé en radians

ATAN (*number*)

Renvoie la tangente inverse de *nombre*, exprimée en radians.

ATAN2 (*x, y*)

Renvoie la tangente inverse du point *x,y*.

CHAR (*code1, [code2, ...]*)

Renvoie une chaîne obtenue par conversion de chaque code ASCII vers le caractère correspondant.

CONCAT (*chaîne1, [chaîne2, ...]*)

Renvoie la concaténation de tous les arguments.

CONV (*nombre, base1, base2*)

Renvoie la conversion de *nombre* de *base1* en *base2*. La base est un chiffre entre 2 et 36.

BIN (*décimal*)

Renvoie la valeur binaire d'un nombre décimal.

BIT_COUNT (*nombre*)

Renvoie le nombre de bits à 1 dans la représentation binaire de *nombre*.

CEILING (*nombre*)

Renvoie le plus petit entier supérieur ou égal à *nombre*.

COS (*radians*)

Renvoie le cosinus de *radians*.

COT (*radians*)

Renvoie la cotangente de *radians*.

CURDATE ()

Renvoie la date courante au format `AAAAMMJJ` ou `AAAA-MM-JJ` selon que le contexte est numérique ou alphanumérique. Forme équivalente : `CURRENT_DATE()`.

CURTIME ()

Renvoie l'heure courante au format `HHMMSS` ou `HH:MM:SS` selon que le contexte est numérique ou alphanumérique. Forme équivalente : `CURRENT_TIME()`.

DATABASE ()

Renvoie le nom de la base de données courante.

DATE_ADD (*date,* `INTERVAL` *durée période*)

Ajoute un nombre *durée* de *période* à *date*. Par exemple `DATE_ADD ("2000-12-01", INTERVAL 3 MONTH)` renvoie "2001-03-01". Les valeurs autorisées pour *période* sont `SECOND`, `MINUTE`, `HOUR`, `DAY`, `MONTH` et `YEAR`. Forme équivalente : `ADDDATE()`.

DATE_FORMAT (`date, format`)

Formate une date d'après un format spécifié avec les options suivantes :

- *%a* Nom court du jour (en anglais : « Mon », « Tue », etc.).
- *%b* Nom court du mois (en anglais : « Jan », « Feb », etc.).
- *%D* Jour du mois avec suffixe (en anglais : « 1st », « 2nd », etc.).
- *%d* Jour du mois.
- *%H* Heure, sur 24 heures, et avec deux chiffres.
- *%h* Heure, sur 12 heures.
- *%i* Minutes.
- *%j* Jour de l'année.
- *%k* Heure, sur 24 heures, et avec un ou deux chiffres.
- *%l* Heure, sur 12 heures, et avec un ou deux chiffres.
- %M Nom du mois (en anglais).
- *%m* Numéro du mois.
- *%p* AM ou PM.
- *%r* Heure complète (HH::MM::SS), sur 12 heures, avec AM ou PM.
- *%S* Secondes, sur deux chiffres.
- *%s* Secondes, sur un ou deux chiffres.
- *%T* Heure complète (HH::MM::SS), sur 24 heures.
- *%U* Numéro de la semaine (N.B. : la semaine commence le dimanche).
- *%W* Nom du jour (en anglais).
- *%w* Numéro du jour de la semaine (NB : la semaine commence le dimanche, jour 0).
- %Y Année sur quatre chiffres.
- %y Année sur deux chiffres.
- %% Pour écrire « % ».

Les quelques fonctions qui suivent fournissent des raccourcis pour des formatages de date courants.

DATE_SUB (`date,INTERVAL durée période`)

Soustrait une duréee à une date. Voir `DATE_ADD()` pour les valeurs des paramètres.

DAYNAME (`date`)

Nom du jour (en anglais).

DAYOFMONTH (`date`)

Numéro du jour dans le mois.

DAYOFWEEK (*date*)

Numéro du jour dans la semaine.

DAYOFYEAR (*date*)

Numéro du jour dans l'année.

DEGREES (*radians*)

Conversion de radians en degrés.

ELT (*nombre, chaîne1, chaîne2, ...*)

Retourne la chaîne dont la position est *nombre*, ou `NULL` si la position n'est pas valide.

ENCRYPT (*chaîne [, clé]*)

Crypte *chaîne*, en utilisant *clé* si ce paramètre est fourni.

FIELD (*chaîne, chaîne1, chaîne2, ...*)

Renvoie la position de la première chaîne identique à *chaîne* parmi {*chaîne1, chaîne2, ...*}, 0 si rien n'est trouvé.

FIND_IN_SET (*chaîne, ensemble*)

Renvoie la position de *chaîne* dans *ensemble*, donné sous la forme `'elem1, elem2, ...'`.

FLOOR (*nombre*)

Plus petit entier inférieur ou égal à *nombre*.

FORMAT (*nombre, décimales*)

Formate un numérique avec un nombre de décimales (NB : l'arrondi à *décimales* est effectué).

FROM_DAYS (*jours*)

Renvoie la date correspondant à *jours*, le jour 1 étant le premier janvier de l'an 1.

FROM_UNIXTIME (*secondes [, format]*)

Renvoie la date GMT correspondant au nombre de secondes écoulées depuis le 1/01/1970 GMT. On peut utiliser un format identique à celui de `DATE_FORMAT()`.

GET_LOCK (*nom, durée*)

Crée un verrou nommé *nom*, actif pendant *durée* secondes. La fonction renvoie 1 si le verrou peut être obtenu, 0 sinon.

GREATEST (*nombre1, nombre2 [, nombre3, ...]*)

Renvoie le nombre le plus grand.

HEX (*décimal*)

Renvoie la valeur hexadécimale de *décimal*.

HOUR (*temps*)

Renvoie l'heure de *temps*. Donc `HOUR('12:10:01')` renvoie 12.

IF (*test, val1, val2*)

Si *test* est vrai, renvoie *val1*, sinon *val2*.

IFNULL (*valeur, valeur2*)

Renvoie *valeur* si *valeur* n'est pas à `NULL`, *valeur2* sinon.

INSERT (*chaîne, position, longueur, chaîne2*)

Renvoie une chaîne obtenue en remplaçant la sous-chaîne de *chaîne* de longueur *longueur* et commençant à *position* par *chaîne2*.

INSTR (*chaîne, souschaîne*)

Renvoie la position de *souschaîne* dans *chaîne*.

ISNULL (*expression*)

Renvoie vrai (1) si *expression* est à `NULL`, 0 sinon.

INTERVAL (*valeur, v1, v2, ...*)

Renvoie 0 si *valeur* est la plus petite valeur, 1 *valeur* est comprise entre *v1* et *v2*, etc.

LAST_INSERT_ID ()

Renvoie la dernière valeur générée pour un champ `AUTO_INCREMENT`.

LCASE (*chaîne*)

Renvoie *chaîne* en minuscules. Synonyme : `LOWER_CASE`.

LEAST (*nombre1, nombre2 [, nombre3, ...]*)

Renvoie le plus petit nombre.

LEFT (*chaîne, longueur*)

Renvoie les *longueur* premiers caractères de *chaîne*.

LENGTH (*chaîne*)

Renvoie la longueur de *chaîne*. Synonymes : `CHAR_LENGTH()`, `CHARACTER_LENGTH()`.

LOCATE (*souschaîne, chaîne [, nombre]*)

Idem que `INSTR()`, mais avec des arguments inversés. *nombre* indique la position à partir de laquelle on recherche la sous-chaîne.

LOG (*nombre*)

Logarithme népérien de *nombre*.

LOG10 (*nombre*)

Logarithme base 10 de *nombre*.

LPAD (*chaîne, longueur, motif*)

Renvoie *chaîne* complétée à gauche avec *motif* autant de fois que nécessaire pour que la longueur soit *longueur*.

LTRIM (*chaîne*)

Retire tous les caractères blancs au début de *chaîne*.

MID (*chaîne, position, longueur*)

Renvoie la sous-chaîne de longueur *longueur* de *chaîne*, débutant à la position *position*. Synonyme : `SUBSTRING()`.

MINUTE (*temps*)

Renvoie les minutes de *temps*. Donc `MINUTE('12:10:01')` renvoie 10.

MOD (*nombre1, nombre2*)

Renvoie *nombre1* modulo *nombre2*.

MONTH (*date*)

Renvoie le mois de *date* (un nombre).

MONTHNAME (*date*)

Renvoie le nom du mois de *date* (en anglais).

NOW ()

Renvoie la date et l'heure courantes. Synonymes : `SYSDATE()` et `CURRENT_TIMESTAMP()`.

OCT (*décimal*)

Renvoie la valeur octale de *décimal*.

PASSWORD (*chaîne*)

Cryptage de *chaîne* avec la fonction utilisée pour les mots de passe MySQL.

PERIOD_ADD (*date, nbMois*)

Ajoute *nbMois* mois à *date* qui doit être au format `AAMM` ou `AAAAMM`.

PERIOD_DIFF (*date2, date2*)

Renvoie le nombre de mois entre les deux dates, qui doivent être au format `AAMM` ou `AAAAMM`.

PI ()

Renvoie le nombre π.

POW (*n1, n2*)

Renvoie $n1^{n2}$. Synonyme : `POWER()`.

QUARTER (*date*)

Renvoie le numéro du trimestre de *date*.

RADIANS (*degré*)

Renvoie l'angle *degré* converti en radians.

RAND (*[gener]*)

Renvoie une valeur aléatoire entre 0 et 1. Le générateur *gener* peut être spécifié optionellement.

RELEASE_LOCK (*nom*)

Relâche le verrou *nom* créé avec la fonction `GET_LOCK()`. Renvoie 1 si l'opération réussit, 0 si elle échoue, et `NULL` si le verrou n'existe pas.

REPEAT (*chaîne, n*)

Renvoie une chaîne constituée de *chaîne* répétée *n* fois.

REPLACE (*chaîne, nouveau, ancien*)

Renvoie une chaîne où toutes les occurrences de *ancien* sont remplacées par *nouveau*.

REVERSE (*chaîne*)

Renvoie la chaîne miroir de *chaîne*.

RIGHT (*chaîne, longueur*)

Renvoie la sous-chaîne de longueur *longueur* à partir de la fin de *chaîne*.

ROUND (*nombre [, nbDec]*)

Arrondit *nombre* au nombre de décimales donné par *nbDec*. Si ce dernier n'est pas spécifié, la fonction arrondit à l'entier le plus proche.

RPAD (*chaîne, longueur, motif*)

Renvoie *chaîne* complétée à droite avec *motif* autant de fois que nécessaire pour que la longueur soit *longueur*.

RTRIM (*chaîne, longueur, motif*)

Renvoie *chaîne* sans les éventuels caractères blancs à sa fin.

SECOND (*temps*)

Renvoie les secondes de *temps*. Donc `SECOND('12:10:01')` renvoie 1.

SEC_TO_TIME (*secondes*)

Renvoie le nombre d'heures, de minutes et de secondes, au format `HH:MM:SS` ou `HHMMSS` selon le contexte, dans *secondes*.

SIGN (*nombre*)

Renvoie le signe de *nombre*.

SIN (*radians*)

Renvoie le sinus de *radians*.

SOUNDEX (*chaîne*)

Renvoie le code Soundex de *chaîne* (utilisé pour les comparaisons de chaînes).

SPACE (*nombre*)

Renvoie une chaîne avec *nombre* caractères blancs.

SQRT (*nombre*)

Renvoie $\sqrt{nombre}$.

STRCMP (*chaîne1, chaîne2*)

Renvoie 0 si les chaînes sont identiques, -1 si *chaîne1* est avant *chaîne2* dans l'ordre lexicographique, 1 sinon.

SUBSTRING_INDEX (*chaîne, car, n*)

Renvoie une sous-chaîne obtenue en comptant *n* fois le caractère *car* dans *chaîne*, en prenant tout ce qui est à gauche si *n* est négatif, tout ce qui est à droite sinon.

SUBSTRING (*chaîne, position, longueur*)

Voir `MID()`.

TAN (*radians*)

Renvoie la tangente de *radians*.

TIME_FORMAT (*temps, format*)

Formate *temps* selon *format*. Voir la fonction DATE_FORMAT().

TIME_TO_SECOND (*temps*)

Renvoie le nombre de secondes dans *temps*.

TO_DAYS (*date*)

Renvoie le nombre de jours entre le 01/01/01 et *date*.

TRIM ([BOTH|LEADING|TRAILING] *[car] [FROM] chaîne*)

Permet de retirer le caractère *car* en début ou en fin de *chaîne*, ou les deux. Par défaut, *car* est le caractère blanc.

TRUNCATE (*nombre, nbDec*)

Tronque *nombre* (sans arrondi !) au nombre de décimales donné par *nbDec*.

UCASE (*chaîne*)

Renvoie *chaîne* en majuscules. Synonyme : UPPER.

UNIX_TIMESTAMP (*[date]*)

Renvoie le nombre de secondes écoulées entre le 01/01/1970 GMT et *date*. Par défaut, *date* est la date courante.

USER ()

Renvoie le nom de l'utilisateur courant. Synonymes : SYSTEM_USER(), SESSION_USER().

VERSION ()

Renvoie la version de MySQL.

WEEK (*date*)

Renvoie le numéro de la semaine de *date*.

YEAR (*date*)

Renvoie l'année de *date*.

C

Fonctions PHP

PHP propose un nombre impressionnant de fonctions pour tous les usages : accès aux fichiers, accès aux bases de données, programmation réseau, production de fichiers PDF, programmation LDAP, etc. Il est évidemment hors de question de les énumérer ici d'autant que vous trouverez facilement, dans la documentation en ligne sur *www.php.net*, une description régulièrement actualisée de toutes les fonctions PHP, accompagnée de commentaires et d'exemples. La documentation PHP peut également être téléchargée librement sur le Web, ce qui permet de disposer au format PDF ou HTML d'une liste des fonctions. Cette liste évolue d'ailleurs rapidement et la documentation en ligne est le meilleur moyen de se tenir à jour.

Comme il est cependant agréable de disposer d'un document récapitulant les fonctions les plus couramment utilisées, cette annexe présente une sélection comprenant les catégories suivantes :

- Page 486 : fonctions « générales », les plus utiles.
- Page 493 : les principales fonctions de manipulation de chaînes de caractères.
- Page 496 : fonctions de manipulation de dates.
- Page 497 : fonctions d'accès aux tableaux.
- Page 504 : fonctions d'accès aux fichiers.
- Page 500 : fonctions XML présentées dans le chapitre 8.
- Page 507 : enfin, bien entendu, la liste complète des fonctions de l'interface PHP/MySQL.

Au sein de chaque groupe, les fonctions sont classées par ordre alphabétique.

Si vous ne trouvez pas, dans la liste qui suit, la fonction d'intérêt général dont vous auriez besoin, ne vous lancez surtout pas dans l'écriture par vous-même de cette fonction avant d'avoir lancé une recherche dans la documentation en ligne très efficace et directe que vous trouverez à l'adresse *www.php.net*.

C.1 FONCTIONS GÉNÉRALES

abs

```
number abs (number nombre)
```

Renvoie la valeur absolue de *nombre*, qui peut être un entier ou un flottant.

basename

```
string basename (string chemin)
```

Étant donné un chemin d'accès à un fichier, la fonction renvoie le nom du fichier. La fonction complémentaire est `dirname()` qui renvoie le chemin sans le nom du fichier.

call_user_func

```
mixte call_user_func (float nomFonction[, mixte param1
                     [, mixte param2]])
```

Appelle la fonction *nomFonction* avec les paramètres *param1*, *param2*.

call_user_method

```
mixte call_user_method (float nomMéthode, objet,
                       [, mixte param1[, mixte param2]])
```

Appelle la méthode *nomMéthode* sur l'objet *objet*. Voir page 339 pour un exemple d'utilisation de cette fonction.

ceil

```
int ceil (float nombre)
```

Renvoie le plus petit entier supérieur ou égal à *nombre*.

dirname

```
string dirname (string chemin)
```

Étant donné un chemin d'accès à un fichier, la fonction renvoie le chemin, sans le nom du fichier.

empty

```
bool empty (variable)
```

Renvoie faux si la variable est définie et a une valeur non nulle, vrai sinon.

eval

```
mixed eval (string commandes)
```

Cette fonction exécute les commandes PHP contenues dans la chaîne de caractères *commandes*. Cela peut être utile pour des applications où des commandes sont créées « à la volée », ou pour exécuter des scripts stockés dans une base de données. Si une des commandes exécutées est `return val`, la valeur *val* est aussi celle qui est renvoyée par `eval()`.

exec

```
string exec (string commande, [, string tableau [, int retour]])
```

Cette fonction exécute une commande système et renvoie la dernière ligne produite par l'exécution de cette commande. Rien n'est donc envoyé au navigateur. Si on passe un paramètre *tableau*, ce dernier contiendra à la fin de l'exécution toutes les lignes produites par *commande*. Si de plus on passe un paramètre *retour*, il contiendra à la fin de l'exécution le code retour de *commande*.

Il est déconseillé d'exécuter des commandes provenant de formulaires. Si c'est nécessaire, la fonction `EscapeShellCmd()` offre un mode plus sécurisé que `exec()`.

floor

```
int floor (float nombre)
```

Renvoie le plus grand entier inférieur ou égal à *nombre*.

define

```
bool define (string nomConstante, mixed valeur
             [, bool insensibleCasse])
```

Définit une constante de nom *nomConstante* avec la valeur *valeur*. Par défaut, le nom d'une constante est sensible à la casse (autrement dit `CONSTANTE` est différent de `Constante`). Ce n'est plus vrai si le dernier argument (optionnel) est à `vrai`.

defined

```
bool defined (string nomConstante)
```

Renvoie `vrai` si la constante *nomConstante* est définie.

ereg

```
int ereg (string motif, string chaîne[, array tabOcc])
```

Évalue une expression régulière et place les occurrences trouvées dans *tabOcc*. Voir page 86 pour un développement sur les expressions régulières.

eregi

```
int eregi (string motif, string chaîne[, array tabOcc])
```

Idem que la précédente, mais l'évaluation n'est pas sensible à la casse (majuscules/-minuscules).

ereg_replace

```
string ereg_replace (string motif, string remplacement,
                     string chaîne)
```

Évalue une expression régulière, remplace les occurrences trouvées par *remplacement*, et renvoie la nouvelle chaîne.

eregi_replace

```
int eregi_replace (string motif, string remplacement,
                   string chaîne)
```

Idem que la précédente, mais l'évaluation n'est pas sensible à la casse (majuscules/-minuscules).

extension_loaded

```
bool extension_loaded (string nomExtension)
```

Renvoie `true` si l'extension est chargée, `false` sinon. Les noms des extensions sont affichés avec `phpinfo()`.

getenv

```
string getenv (string variable)
```

Renvoie la valeur de la variable d'environnement *variable*, ou 0 si elle n'existe pas.

getType

```
string getType (variable)
```

Renvoie le type d'une variable.

Header

```
Header (chaîne)
```

Cette fonction produit un en-tête HTTP comme `Content-type`, `Location`, `Expires`, etc. Elle doit être utilisée avant toute production de texte HTML.

htmlEntities

```
string htmlEntities (string chaîne)
```

Cette fonction remplace tous les caractères réservés de HTML (comme par exemple « & »), par une entité (par exemple `&`).

is_array

```
bool is_array (variable)
```

Renvoie vrai si *variable* est un tableau.

is_double

```
bool is_double (variable)
```

Renvoie vrai si *variable* est de type `double`.

is_float

```
bool is_float (variable)
```

Renvoie vrai si *variable* est de type `float`.

is_int

```
bool is_int (variable)
```

Renvoie vrai si *variable* est un entier.

is_long

```
bool is_long (variable)
```

Renvoie vrai si *variable* est de type `long`.

is_object

```
bool is_object (variable)
```

Renvoie vrai si *variable* est un objet.

is_string

```
bool is_string (variable)
```

Renvoie vrai si *variable* est une chaîne.

isSet

```
bool isSet (variable)
```

Renvoie vrai si la variable est définie et a une valeur, faux sinon.

mail

```
bool mail (string destinataire,
           string sujet,
           string texte,
           string ajoutEntête)
```

Cette fonction envoie un email. Le dernier paramètre, optionnel, permet d'ajouter des informations dans l'en-tête de l'email.

max

```
mixed max (mixed var1, mixed var2, ... mixed varN)
```

Renvoie la plus grande des variables passées en argument. On peut aussi lui passer un tableau, au lieu d'une liste de variables.

md5

```
string md5 (string chaîne)
```

MD5 est une fonction de hachage qui renvoie une chaîne de 32 octets associée à la chaîne *chaîne*. Il est à peu près impossible d'obtenir deux valeurs identiques pour des chaînes différentes (collision) ce qui permet de considérer cette fonction comme un cryptage de chaîne.

min

```
mixed min (mixed var1, mixed var2, ... mixed varN)
```

Renvoie la plus petite des variables passées en argument. On peut aussi passer un tableau, au lieu d'une liste de variables.

nl2br

```
string nl2br (string chaîne)
```

Remplace les retours à la ligne dans *chaîne* par <br/> pour que ces retours soient reportés correctement dans un affichage HTML.

passthru

```
string passthru (string commande [, int retour]])
```

Cette fonction est identique à `exec()`, mais le résultat produit par *`commande`* est envoyé directement au navigateur. En d'autres termes, on fait appel à un programme ou à une commande système auxiliaire pour produire le document, ou l'extrait de document, transmis au navigateur. La fonction `system()` est analogue à `passthru()`.

phpInfo

```
phpInfo ()
```

Affiche des informations sur l'interpréteur PHP.

print

```
print (string chaîne)
```

Imprime une chaîne sur la sortie standard. Identique à `echo()` – à ceci près qu'il faut encadrer *`chaîne`* par des parenthèses.

print_r

```
print_r (array tab)
```

Affiche tous les éléments d'un tableau (fonctionne aussi pour n'importe quel type de variable).

printf

```
printf (string format, variable1, variable2, ..)
```

Imprime une liste de variables selon un format décrit comme pour la fonction C de même nom.

putenv

```
putenv (string expression)
```

Définit une variable d'environnement comme indiqué dans *`expression`* qui peut être, par exemple, `"PATH=$path"`.

rand

```
int rand ()
```

Engendre une valeur aléatoire.

round

```
int round (float nombre)
```

Renvoie l'entier le plus proche de *nombre*.

SetCookie

```
bool setCookie (string nom,
                string valeur,
                int expiration,
                string chemin,
                string domaine,
                string sécurité)
```

Cette fonction permet de demander au navigateur d'enregistrer un *cookie* avec l'en-tête `Set-Cookie` : voir page 17. Tous les paramètres sauf le premier sont optionnels : si seul le nom est indiqué, le *cookie* sera supprimé. Les *cookies* doivent faire partie de l'en-tête d'un document, ce qui implique que la fonction doit être appelée avant de produire la première balise HTML.

La durée de vie du *cookie* est indiquée par *expiration* exprimée en temps Unix. On peut donc appeler la fonction `time()` et ajouter au résultat le nombre de secondes donnant la durée de vie. Par défaut un *cookie* disparaît quand le programme client (navigateur) s'arrête. Les paramètres *chemin* et *domaine* définissent la visibilité du *cookie* pour les serveurs qui dialoguent par la suite avec le navigateur (voir page 17). Par défaut, seul le serveur qui a créé le *cookie* peut le lire. Enfin, *sécurité* indique que le *cookie* sera transféré en mode sécurisé avec SSL (*Secure Socket Layer*).

unset

```
unset (variable)
```

Détruit *variable*, qui n'est alors plus définie.

urlDecode

```
string urlDecode (string chaîne)
```

Cette fonction décode le paramètre *chaîne* provenant d'une URL, pour y remplacer par exemple les « + » par des espaces.

urlEncode

```
string urlEncode (string chaîne)
```

Cette fonction code le paramètre *chaîne* de manière à ce qu'elle puisse être transmise dans une URL. Les blancs dans *chaîne* sont par exemple transformés en « + » dans la chaîne produite.

C.2 CHAÎNES DE CARACTÈRES

addSlashes

```
string addSlashes (string chaîne)
```

Renvoie une chaîne identique à ***chaîne***, avec des « \ » devant chaque apostrophe simple (’), apostrophe double (") et barre oblique inversée (\). La chaîne ainsi obtenue peut être utilisée sans risque dans une requête SQL.

chop

```
string chop (string chaîne)
```

Renvoie une chaîne identique à ***chaîne***, après suppression de tous les caractères blancs en fin de chaîne.

explode

```
array explode (string séparateur, string chaîne)
```

Divise ***chaîne*** en valeurs séparées par ***séparateur***, et renvoie le tableau de ces valeurs.

implode

```
string implode (array tableau, string séparateur)
```

Fonction inverse de `explode()` : renvoie une chaîne avec les valeurs de ***tableau*** séparées par ***séparateur***.

ltrim

```
string ltrim (string chaîne)
```

Renvoie une chaîne identique à ***chaîne***, après suppression de tous les caractères blancs en début de chaîne.

strchr

```
string strchr (string chaîne1, string chaîne2)
```

Renvoie le contenu de ***chaîne1*** à partir de la première occurrence de ***chaîne2***. Renvoie faux si ***chaîne2*** n’apparaît pas dans ***chaîne1***.

strcmp

```
int strcmp (string chaîne1, string chaîne2)
```

Renvoie une valeur négative si *chaîne1* précède *chaîne2* dans l'ordre lexicographique, une valeur positive si *chaîne1* est supérieure à *chaîne2*, 0 si les chaînes sont égales.

strcspn

```
int strcspn (string chaîne1, string chaîne2)
```

Renvoie la position du premier caractère de *chaîne1* qui fait aussi partie de *chaîne2*.

stripSlashes

```
string stripSlashes (string chaîne)
```

Renvoie une chaîne identique à *chaîne*, avec suppression des « \ » devant chaque apostrophe simple (’), apostrophe double (") et barre oblique inversée (\). C'est la fonction inverse de `addSlashes()`.

strlen

```
int strlen (string chaîne)
```

Renvoie la longueur de *chaîne*.

strpos

```
int strpos (string chaîne1, string chaîne2)
```

Renvoie la position de la première occurrence de *chaîne2* dans *chaîne1*. Renvoie faux si rien n'est trouvé.

strrpos

```
int strrpos (string chaîne, char caractère)
```

Renvoie la position de la dernière occurrence de *caractère* dans *chaîne*. Renvoie faux si rien n'est trouvé.

substr_count

```
int substr_count (string chaîne1, string chaîne2)
```

Renvoie le nombre d'occurrences de *chaîne2* dans *chaîne1*

strrchr

```
string strrchr (string chaîne1, string chaîne2)
```

Renvoie le contenu de *chaîne1* à partir de la dernière occurrence de *chaîne2*. Renvoie faux si *chaîne2* n'apparaît pas dans *chaîne1*.

strrev

```
string strrev (string chaîne)
```

Renvoie la chaîne miroir de *chaîne*.

strspn

```
int strspn (string chaîne1, string chaîne2)
```

Renvoie la position du premier caractère de *chaîne1* qui ne fait pas partie de *chaîne2*.

strToLower

```
string strToLower (string chaîne)
```

Renvoie la chaîne mise en minuscules.

strToUpper

```
string strToUpper (string chaîne)
```

Renvoie la chaîne mise en majuscules.

substr

```
string substr (string chaîne, int début, int longueur)
```

Renvoie la sous-chaîne de *chaîne* de longueur *longueur* à partir de *début*.

trim

```
string trim (string chaîne)
```

Renvoie une chaîne identique à *chaîne* après suppression des blancs au début et à la fin.

C.3 DATES

checkdate

```
bool checkdate (int mois, int jour, int an)
```

Contrôle la validité d'une date : le 32/89/1879879 n'est pas une date valide !

date

```
string date (string format [, int timestamp])
```

Renvoie la date correspondant à *timestamp*, ou la date courante si le paramètre est omis, formaté avec *format*. Les options pour le formatage sont les suivantes.

- *a* « am » ou « pm ».
- *A* « AM » ou « PM ».
- *d* Jour du mois, sur deux chiffres avec un zéro initial si besoin est.
- *D* Trois premières lettres du nom du jour (en anglais).
- *F* Nom du mois (en anglais).
- *h* Heure, sur 12 heures, avec deux chiffres.
- *H* Heure, sur 24 heures, et avec deux chiffres.
- *g* Heure, sur 12 heures, avec un ou deux chiffres.
- G Heure, sur 24 heures, et avec un ou deux chiffres.
- *i* Minutes, de 00 à 59.
- *j* Jour du mois, sur un ou deux chiffres.
- *l* Nom du jour (en anglais).
- *L*, 0 ou 1 selon qu'il s'agit d'une année bissextile ou pas.
- *m* Numéro du mois, de 01 à 12.
- *n* Numéro du mois, de 1 à 12.
- M Trois premières lettres du nom du mois (en anglais).
- *s* Secondes, de 00 à 59.
- *t* Nombre de jours dans le mois courant (28 à 31).
- *U* Nombre de secondes depuis le 01/01/1970.
- *w*, chiffre du jour de la semaine, de 0 (dimanche) à 6 (samedi).
- *Y* Année sur quatre chiffres.
- *y* Année sur deux chiffres.
- *z* Numéro du jour de l'année, commençant à 0.

getdate

```
array getdate (int timestamp)
```

Renvoie les informations propres à *timestamp* sous la forme d'un tableau associatif contenant des éléments indexés par les clés suivantes :

- *seconds* : les secondes.
- *minutes* : les minutes.
- *hours* : les heures.
- *mday* : jour du mois.
- *wday* : numéro du jour de la semaine.
- *mon* : numéro du mois.
- *year* : l'année.
- *yday* : numéro du jour dans l'année.
- *weekday* : nom (en anglais) du jour.
- *month* : nom (en anglais) du mois.

`mkTime`

```
int mkTime (int heure, int minutes, int secondes,
            int mois, int jour, int annee)
```

Renvoie un *timestamp* UNIX (secondes depuis le 01/01/1970).

`time`

```
int time ()
```

Renvoie le *timestamp* UNIX (secondes depuis le 01/01/1970) de la date courante.

C.4 TABLEAUX

Rappelons qu'un tableau est une suite de *valeurs*, indexées par un chiffre ou par une *clé* (tableaux associatifs). Le terme « élément » désigne la paire clé/valeur ou indice/-valeur. La liste qui suit est une sélection qui ne donne pas de manière exhaustive les fonctions d'interaction avec les tableaux PHP.

`array`

```
array array (listeValeurs)
```

Crée un tableau avec initialisation à partir de la liste des valeurs fournies. Voir le chapitre 11 pour l'utilisation de cette fonction.

`array_key_exists`

```
bool array_key_exists (clé, tableau)
```

Renvoie `true` si la clé existe dans le tableau.

arsort

```
arsort (array tableau)
```

Trie le tableau associatif *tableau* sur les valeurs, en ordre descendant, et en gardant l'association clé/valeur.

asort

```
asort (array tableau)
```

Trie le tableau associatif *tableau* sur les valeurs, en ordre ascendant, et en gardant l'association clé/valeur.

count

```
int count (array tableau)
```

Renvoie le nombre d'éléments du tableau.

current

```
string current (array tableau)
```

Chaque tableau dispose d'un pointeur interne qui adresse, initialement, le premier élément. Cette fonction renvoie la valeur de l'élément courant du tableau, sans modifier le pointeur interne.

each

```
array each (array tableau)
```

Cette fonction renvoie la clé et la valeur de l'élément courant, et avance le curseur. Le résultat est un tableau à quatre éléments, avec les clés 0, 1, `key` et `value`. On peut typiquement exploiter ce résultat avec la construction `list()`.

```
while ( list ($cle, $element) = each ($tableau))
  ...
```

end

```
end (array tableau)
```

Positionne le pointeur interne sur le dernier élément du tableau.

in_array

```
bool in_array (valeur, tableau)
```

Renvoie `true` si la valeur existe dans le tableau.

key

```
mixed key (array tableau)
```

Renvoie la clé de l'élément courant du tableau, sans modifier le pointeur interne.

ksort

```
ksort (array tableau)
```

Trie le tableau associatif sur la clé.

list

```
list (variable1, variable2, ... variableN = tableau)
```

Il s'agit d'une construction syntaxique plus que d'une fonction. Elle permet d'affecter en une seule instruction la valeur des variables *variable1*, *variable2*, ... *variableN* avec les *N* premiers éléments d'un tableau. Voir l'exemple donné pour la fonction `each()`.

max

```
max (array tableau)
```

Renvoie la plus grande valeur du tableau.

min

```
min (array tableau)
```

Renvoie la plus petite valeur du tableau.

next

```
mixed next (array tableau)
```

Renvoie la valeur du prochain élément du tableau et avance le pointeur interne. La fonction renvoie faux quand le dernier élément est dépassé.

prev

```
mixed prev (array tableau)
```

Renvoie la valeur de l'élément précédent du tableau et recule le pointeur interne. La fonction renvoie faux quand le premier élément est dépassé.

reset

```
reset (array tableau)
```

Positionne le pointeur interne sur le premier élément du tableau.

rsort

```
rsort (array tableau)
```

Trie le tableau sur les valeurs, en ordre descendant.

sort

```
sort (array tableau)
```

Trie le tableau sur les valeurs, en ordre ascendant.

C.5 FONCTIONS XML

Les fonctions XML données ci-dessous sont celles de l'interface *SimpleXML*, puis de l'API SAX. L'utilisation de ces fonctions est décrite dans le chapitre 8 où se trouvent également les quelques fonctions du processeur XSLT. On trouve dans les versions récentes de PHP (postérieures à la 4.2.0) des fonctions correspondant à l'interface DOM de traitement des documents XML, non décrites ici.

SimpleXML

Les trois fonctions suivantes permettent de créer un objet SimpleXML. L'objet obtenu représente l'élément racine du document.

SimpleXML_load_file

```
object SimpleXML_load_file (string nomFichier)
```

Charge le document XML du fichier *nomFichier* dans un objet SimpleXML.

SimpleXML_load_string

```
object SimpleXML_load_string (string chaîne)
```

Charge le document XML contenu dans *chaîne* dans un objet SimpleXML.

SimpleXML_import_DOM

```
object SimpleXML_import_DOM (object nœud)
```

Charge le document XML à partir du nœud d'un document DOM.

Les fonctions qui suivent sont des méthodes applicables à un objet SimpleXML, désigné par « objet-cible ». Voir page 333.

asXML

```
string asXML ()
```

Renvoie une chaîne contenant la représentation sérialisée du document XML stocké dans l'objet-cible.

attributes

```
tableau attributes ()
```

Renvoie un tableau associatif contenant les attributs (nom et valeur) de l'objet-cible.

children

```
tableau children ()
```

Renvoie un tableau d'objets contenant les éléments-fils de l'objet-cible.

XPath

```
tableau XPath (chaîne expressionXPath)
```

Renvoie un tableau d'objets contenant le résultat de l'évaluation de l'expression XPath en prenant l'objet-cible comme nœud contexte.

SAX

xml_error_string

```
string xml_error_string (int code)
```

Renvoie le message d'erreur correspondant à la valeur de *code*, ce dernier étant fourni par la fonction `xml_get_error_code()`.

xml_get_current_byte_index

```
int xml_get_current_byte_index (parseur)
```

Renvoie la position courante du parseur au cours de son analyse.

xml_get_current_column_number

```
int xml_get_current_column_number (parseur)
```

Renvoie la position du parseur dans la ligne courante. Utile pour retrouver l'emplacement d'une erreur.

xml_get_current_line_number

```
int xml_get_current_line_number (parseur)
```

Renvoie la ligne courante où se trouve le parseur.

xml_get_error_code

```
int xml_get_error_code (parseur)
```

Renvoie le code de la dernière erreur rencontrée.

xml_parse

```
xml_parse (parseur, données, [final])
```

Analyse le fragment de document XML contenu dans le paramètre *données*. Ce fragment peut être incomplet tant que *final* vaut `false`, le parseur attendant alors le fragment suivant pour poursuivre son analyse. La valeur `true` pour *final* indique que *données* contient le dernier fragment (ou tout le document). Ce découpage permet d'analyser un document par « paquets » d'une taille raisonnable.

xml_parser_create

```
mixte xml_parser_create ([codage])
```

Crée un parseur. Le paramètre (optionel) *codage* indique le codage des caractères à employer : `ISO-8859-1` (le défaut), `US-ASCII` ou `UTF-8`.

xml_parser_free

```
xml_parser_free (parseur)
```

Détruit un parseur.

xml_parser_get_option

```
string xml_parser_get_option (parseur, string option)
```

Renvoie la valeur de l'option *option*. Seules deux options sont proposées :

- `XML_OPTION_CASE_FOLDING` : booléen indiquant que les noms d'éléments sont mis en majuscules.
- `XML_OPTION_TARGET_ENCODING` : le jeu de caractères pour le codage du résultat.

xml_parser_set_option

```
xml_parser_set_option (parseur, string option, string valeur)
```

Affecte une valeur à une option (voir ci-dessus).

xml_parse_into_struct

```
xml_parse_into_struct (parseur, string donnees,
                       string tabValeur, string tabIndex)
```

Analyse complètement un document XML et le représente dans deux tableaux PHP. Assez complexe : voir la documentation PHP pour un exemple complet.

xml_set_character_data_handler

```
xml_set_character_data_handler (parseur, fChar)
```

Affecte la fonction `fChar()` au traitement des données caractères. Cette fonction doit accepter deux arguments : le parseur, et la chaîne contenant les données caractères.

xml_set_default_handler

```
xml_set_default_handler (parseur, fDef)
```

Affecte la fonction `fDef()` (mêmes arguments que la précédente) au traitement des composants du document XML pour lesquels il n'existe pas d'autre « déclencheur ».

xml_set_element_handler

```
xml_set_element_handler (parseur, fElemDebut, fElemFin)
```

Affecte la fonction `fElemDebut()` au traitement des balises ouvrantes, et `fElemFin()` au traitement des balises fermantes. Voir les exemples du chapitre 8.

xml_set_external_entity_ref_handler

```
xml_set_external_entity_ref_handler (parseur, fEntExt)
```

Affecte la fonction `fEntExt()` au traitement des entités externes.

`xml_set_notation_decl_handler`

xml_set_notation_decl_handler (*parseur*, *fNotation*)

Affecte la fonction `fNotation()` au traitement des notations XML. Les « notations » (très rarement utilisées) fournissent un moyen de décrire dans un document XML des données non alphanumériques.

`xml_set_object`

xml_set_object (*parseur*, *objet*)

Indique au parseur que les déclencheurs sont les méthodes de *objet*.

`xml_set_processing_instruction_handler`

xml_set_processing_instruction_handler (*parseur*, *fPI*)

Affecte la fonction `fPI()` au traitement des instructions de traitement. Cette fonction doit accepter trois paramètres : le parseur, un paramètre *nom* contenant le nom de l'instruction, et un paramètre *données* contenant la chaîne constituant l'instruction.

C.6 ACCÈS AUX FICHIERS

`chdir`

chdir (string *répertoire*)

Permet de se positionner dans *répertoire*.

`closedir`

closedir (int *répertoire*)

Referme le pointeur de répertoire (qui doit avoir été ouvert par `opendir()` auparavant).

`copy`

copy (string *source*, string *destination*)

Copie un fichier de *source* vers *destination*.

`fclose`

fclose (int *descFichier*)

Ferme le fichier identifié par *descFichier*.

feof

```
bool feof (int descFichier)
```

Renvoie `vrai` si la fin du fichier identifié par *descFichier* est atteinte.

fgetc

```
char fgetc (int descFichier)
```

Renvoie le caractère placé à la position courante du fichier identifié par *descFichier*, et avance d'une position.

fgets

```
string fgets (int descFichier, int longueur)
```

Renvoie une ligne du fichier, de taille maximale *longueur*.

file

```
array file (string nomFichier)
```

Charge tout le contenu de *nomFichier* dans un tableau, avec un élément par ligne.

file_exists

```
bool file_exists (string nomFichier)
```

Teste l'existence de *nomFichier*.

filesize

```
int filesize (string nomFichier)
```

Renvoie la taille de *nomFichier*.

fopen

```
int fopen (string nomFichier, string mode)
```

Ouvre le fichier *nomFichier* et renvoie un descripteur qui peut être utilisé ensuite pour lire le contenu. Les modes d'ouverture sont les suivants :

1. *r* : lecture seule.
2. *w* : écriture seule. Le contenu du fichier est effacé s'il existe déjà.
3. *a* : ajout. Le fichier est créé s'il n'existe pas.

On peut combiner des options, comme par exemple `rw` qui indique à la fois lecture et écriture. On peut aussi, dans tous les cas, ajouter un `b` à la fin du mode pour indiquer que le fichier est au format binaire.

Le paramètre *nomFichier* peut être une URL complète, commençant par *http://* ou *ftp://*, suivie du serveur, du chemin d'accès et du nom du fichier proprement dit.

fpassthru

```
int fpasstrhu (int descFichier)
```

Cette fonction permet de transférer directement le contenu d'un fichier – en partant de la position courante dans le fichier– vers le programme client. Elle renvoie le nombre d'octets lus. Le fichier doit avoir été ouvert par `fopen()` auparavant. La fonction `readfile()` est encore plus simple à utiliser puisqu'on se contente de donner le nom du fichier.

fputs

```
bool fputs (int descFichier, string chaîne)
```

La fonction écrit *chaîne* dans le fichier décrit par *descFichier*. Elle renvoie `vrai` si l'opération réussit.

include

```
include (string nomFichier)
```

La fonction ouvre le fichier et exécute les instructions qui y sont contenues. Pour des raisons expliquées page 60, ce mode d'appel à des instructions extérieures au script est à utiliser avec précaution. On peut avantageusement le remplacer par un appel de fonction.

mkdir

```
bool mkdir (string nomRépertoire, string permissions)
```

Crée un répertoire de nom *nomRépertoire*. Les permissions sont codées comme sous UNIX.

opendir

```
opendir (string nomRépertoire)
```

Crée un pointeur de répertoire qui permet de parcourir la liste des fichiers du répertoire avec `readdir()`.

readdir

```
readdir (int répertoire)
```

Retourne le nom du fichier suivant dans le répertoire identifié par *répertoire* (ouvert avec `opendir()`).

readfile

```
int readfile (string nomFichier [, bool chercherPartout])
```

Cette fonction transfère directement le contenu de *nomFichier* vers le programme client et renvoie le nombre d'octets lus. Si le second paramètre est `true`, le fichier est recherché dans tous les répertoires de l'option `include_path` dans le fichier *php.ini*.

rename

```
bool rename (string source, string destination)
```

Renomme le fichier *source* en *destination*.

require

```
require (string nomFichier)
```

La fonction insère le contenu du fichier *nomFichier* dans le script courant. Pour des raisons expliquées page 60, le fichier inclus doit contenir des déclarations de constantes ou de fonctions, mais il vaut mieux éviter d'y placer des instructions ou des définitions de variables.

rmdir

```
bool rmdir (string nomRépertoire)
```

Détruit le répertoire de nom *nomRépertoire*.

C.7 INTERFACE PHP/MySQL

mysql_affected_rows

```
int mysql_affected_rows (int [connexion])
```

mysql_affected_rows renvoie le nombre de lignes modifiées, détruites ou insérées dans une table après une requête `UPDATE`, `DELETE` ou `INSERT`. L'argument `connexion` est optionnel : par défaut, la dernière connexion établie avec MySQL est utilisée.

mysql_change_user

```
int mysql_change_user (string nom,
                       string motPasse,
                       string [base],
                       int [connexion])
```

mysql_change_user permet de modifier le compte de connexion à MySQL. Les arguments sont les suivants :

1. *nom* est le nom de l'utilisateur MySQL.
2. *motPasse* est le mot de passe.
3. *base* est le nom de la base sous laquelle on souhaite travailler après changement d'identité.
4. *connexion* est la connexion pour laquelle on souhaite changer l'utilisateur. Par défaut la dernière connexion ouverte est utilisée.

La fonction renvoie, en cas de succès, un entier positif. Si la connexion échoue, les anciens paramètres de connexion restent valides. Cette fonction est relativement récente (MySQL 3.23.3).

mysql_client_encoding

```
string mysql_client_encoding (int [connexion])
```

renvoie le jeu de caractères de la connexion courante.

mysql_close

```
int mysql_close (int [connexion])
```

`mysql_close()` ferme une connexion avec MySQL. L'argument *connexion* est optionnel : par défaut la dernière connexion ouverte est utilisée. Cette fonction est en général inutile puisqu'une connexion (non persistante) avec MySQL est fermée à la fin du script.

mysql_connect

```
int mysql_connect (string [chaine_connexion],
                   string [nom],
                   string [motPasse])
```

`mysql_connect()` établit une connexion avec MySQL.

1. *chaine_connexion* est au format `[hôte[:port][:fichierSocket]]`. La chaîne `hôte` vaut par défaut `localhost`, le port étant le port par défaut du serveur MySQL. Le chemin d'accès au fichier *socket* peut également être indiqué (à la place du port) pour une connexion à partir de la même machine que celle où tourne le serveur.
2. *nom* est le nom de l'utilisateur MySQL. Par défaut il prend la valeur de l'utilisateur sous lequel le serveur Apache a été lancé (typiquement `nobody`).
3. *motPasse* est le mot de passe. Par défaut, un mot de passe vide est utilisé.

La fonction renvoie, en cas de succès, un entier positif qui est utilisé pour identifier la connexion lors des accès ultérieurs. Notez que la connexion est automatiquement fermée à la fin du script.

`mysql_create_db`

```
int mysql_create_db (string nomBase,
                     int [connexion])
```

`mysql_create_db()` crée une nouvelle base de nom *`nomBase`*, en utilisant la connexion *`connexion`* (ou, par défaut, la dernière connexion ouverte). Bien entendu le compte utilisateur associé à la connexion doit avoir les droits MySQL suffisants.

`mysql_data_seek`

```
int mysql_data_seek (int résultat,
                     int noLigne)
```

`mysql_data_seek()` positionne le curseur sur la ligne *`noLigne`*. La ligne peut alors être récupérée avec une des fonctions `mysql_fetch_***()`. La fonction renvoie `true` si l'opération réussit, et `false` sinon. Les numéros de ligne commencent à 0.

`mysql_db_name`

```
int mysql_db_name (int résultat,
                   int [noLigne])
```

Cette fonction renvoie le nom d'une base de données en prenant en argument un identifiant de résultat fourni par `mysql_list_dbs()`, et le numéro de la ligne souhaitée (`mysql_num_rows()` permet de connaître le nombre de lignes). Elle renvoie `false` en cas d'erreur.

`mysql_db_query`

```
int mysql_db_query (string nomBase,
                    string requête,
                    int [connexion])
```

Cette fonction se positionne sur la base *`nomBase`*, puis exécute la requête *`requête`* en utilisant la connexion *`connexion`* (ou, par défaut, la dernière connexion ouverte). Elle renvoie un identifiant de résultat.

`mysql_drop_db`

```
int mysql_drop_db (string nomBase,
                   int [connexion])
```

Cette fonction tente de détruire la base *nomBase*, en utilisant la connexion *connexion* (ou, par défaut, la dernière connexion ouverte). Bien entendu, le compte utilisateur associé à la connexion doit avoir les droits MySQL suffisants.

mysql_errno

```
int mysql_errno (int [connexion])
```

Renvoie le numéro de l'erreur survenue lors de la précédente opération MySQL – 0 s'il n'y a pas eu d'erreur.

mysql_error

```
string mysql_error (int [connexion])
```

Renvoie le texte de l'erreur survenue lors de la précédente opération MySQL.

mysql_fetch_array

```
array mysql_fetch_array (int résultat,
                         int [typeResultat])
```

Renvoie un tableau associatif contenant les attributs de la ligne courante, et positionne le curseur sur la ligne suivante. Chaque champ du tableau est indexé par le nom de l'attribut correspondant dans la clause `SELECT` de la requête SQL. La fonction renvoie `false` quand il n'y a plus de ligne.

résultat est un identifiant de résultat, renvoyé par une fonction comme *mysql_query*, et *typeResultat* est une constante qui peut prendre les valeurs suivantes :

1. `MYSQL_ASSOC`
2. `MYSQL_NUM`
3. `MYSQL_BOTH` (par défaut)

mysql_fetch_assoc

```
array mysql_fetch_assoc (int résultat)
```

Donne le même résultat que l'appel à `mysql_fetch_array()` avec le type `MYSQL_ASSOC`.

mysql_fetch_field

```
object mysql_fetch_field (int résultat,
                          int [noAttribut])
```

Renvoie un objet donnant des informations sur l'attribut *noAttribut* du résultat identifié par *résultat*. Si *noAttribut* n'est pas spécifié, la fonction accède au prochain attribut parmi ceux qui n'ont pas encore été consultés. Notez que les attributs sont numérotés à partir de 0.

L'objet renvoyé par la fonction contient les informations suivantes :

1. `name`, nom de l'attribut.
2. `table`, nom de la table à laquelle appartient l'attribut.
3. `max_length`, longueur maximale.
4. `not_null`, 1 si l'attribut ne peut être à `NULL`.
5. `primary_key`, 1 si l'attribut est une clé primaire.
6. `unique_key`, 1 si l'attribut est une clé unique.
7. `multiple_key`, 1 si l'attribut est une clé non unique.
8. `numeric`, 1 si l'attribut est un numérique.
9. `blob`, 1 si l'attribut est un `BLOB`.
10. `type`, le type de l'attribut.
11. `unsigned`, 1 si l'attribut est un non signé.
12. `zerofill`, 1 si l'attribut est déclaré avec l'option `ZEROFILL`.

mysql_fetch_lengths

```
array mysql_fetch_lengths (int résultat)
```

Renvoie un tableau indicé (à partir de 0) donnant la longueur de chaque attribut dans la ligne ramenée par le précédent appel à une fonction `mysql_fetch_***()`.

mysql_fetch_object

```
object mysql_fetch_object() (int résultat,
                             int [typeResultat])
```

Renvoie un objet dont chaque propriété correspond à l'un des attributs de la ligne courante et positionne le curseur sur la ligne suivante. Le nom de chaque propriété de l'objet est le nom de l'attribut correspondant dans la clause `SELECT` de la requête SQL. La fonction renvoie `false` quand il n'y a plus de ligne. Les arguments sont identiques à ceux de la fonction *mysql_fetch_array*.

mysql_fetch_row

```
array mysql_fetch_row (int résultat)
```

Renvoie un tableau indicé contenant les attributs de la ligne courante, et positionne le curseur sur la ligne suivante. Les champs sont numérotés à partir de 0. Le paramètre *résultat* est un identifiant de résultat.

mysql_field_flags

```
string mysql_field_flags (int résultat,
                          int noAttribut)
```

Renvoie une chaîne contenant les options de l'attribut *noAttribut* dans le résultat identifié par *résultat*. Ces options sont celles données au moment de la création de la table : `not_null`, `primary_key`, `unique_key`, `multiple_key`, `blob`, `unsigned`, `zerofill`, `binary`, `enum`, `auto_increment`, `timestamp`. Elles apparaissent dans la chaîne séparées par des blancs. La fonction PHP *explode* permet de placer ces valeurs dans un tableau associatif.

mysql_field_len

```
int mysql_field_len (int résultat,
                     int noAttribut)
```

Renvoie la longueur de l'attribut *noAttribut* dans le résultat identifié par *résultat*.

mysql_field_name

```
string mysql_field_name (int résultat,
                         int noAttribut)
```

Renvoie le nom de l'attribut indexé par *noAttribut* dans le résultat identifié par *résultat*. Les attributs sont numérotés à partir de 0. Donc l'appel `mysql_field_name ($result, 2)` renvoie le nom du troisième attribut.

mysql_field_seek

```
int mysql_field_seek (int résultat,
                      int noAttribut)
```

Permet de positionner le curseur sur l'attribut *noAttribut* de la ligne courante. Le prochain appel à la fonction `mysql_fetch_field()`, *sans utiliser le deuxième argument*, ramènera les informations sur cet attribut.

mysql_field_table

```
string mysql_field_table (int résultat,
                          int noAttribut)
```

Renvoie le nom de la table à laquelle appartient l'attribut *noAttribut* dans le résultat identifié par *résultat*.

mysql_field_type

```
string mysql_field_type (int résultat,
                         int noAttribut)
```

Renvoie le type de l'attribut *noAttribut* dans le résultat identifié par *résultat*.

mysql_free_result

```
int mysql_free_result (int résultat)
```

Libère la mémoire affectée au résultat identifié par *résultat*. Cette mémoire est de toute manière libérée à la fin du script, mais la fonction peut être utile si on souhaite récupérer de la mémoire au cours de l'exécution d'un script volumineux.

mysql_get_client_info

```
string mysql_get_client_info (int [connexion])
```

Renvoie la version du client MySQL.

mysql_get_host_info

```
string mysql_get_host_info (int [connexion])
```

Renvoie des informations sur la machine sur laquelle tourne le serveur MySQL.

mysql_get_proto_info

```
string mysql_get_proto_info (int [connexion])
```

Renvoie le protocole de connexion client/serveur

mysql_get_server_info

```
string mysql_get_server_info (int [connexion])
```

Renvoie la version du serveur MySQL

mysql_info

```
string mysql_info (int [connexion])
```

Renvoie une chaîne contenant des informations sur la dernière requête exécutée.

mysql_insert_id

```
int mysql_insert_id ()
```

Renvoie l'identifiant engendré pendant le dernier ordre `INSERT` pour l'attribut bénéficiant de l'option `AUTO_INCREMENT`. Cet identifiant permet de réaccéder (avec un ordre `SELECT`) à une ligne que l'on vient de créer dans une table disposant d'un identifiant automatiquement incrémenté.

mysql_list_dbs

```
int mysql_list_dbs (int [connexion])
```

La fonction renvoie un identifiant qui peut être utilisé par la fonction `mysql_tablename()` ou la fonction `mysql_db_name()` pour obtenir la liste des bases sur le serveur MySQL.

mysql_list_fields

```
int mysql_list_fields (string nomBase,
                       string nomTable,
                       int [connexion])
```

Cette fonction permet d'inspecter la définition de la table *nomTable* dans la base *nomBase*, le paramètre *connexion* ayant la signification habituelle. Elle renvoie un identifiant qui peut être utilisé par les fonctions *mysql_field_flags*, `mysql_field_len()`, `mysql_field_name()`, et `mysql_field_type()`. En cas d'erreur, la fonction renvoie -1.

Une autre possibilité est d'effectuer un appel `mysql_query("SHOW COLUMNS FROM nomTable");` qui renvoie un tableau décrivant les champs.

mysql_list_tables

```
int mysql_list_tables (string nomBase,
                       int [connexion])
```

La fonction renvoie un identifiant qui peut être utilisé par la fonction `mysql_tablename()` pour obtenir la liste des tables de la base *nomBase*.

mysql_num_fields

```
int mysql_num_fields (int résultat)
```

La fonction renvoie le nombre d'attributs dans les lignes du résultat identifié par *résultat*.

mysql_num_rows

```
int mysql_num_rows (int résultat)
```

La fonction renvoie le nombre de lignes dans le résultat identifié par *résultat*.

mysql_pconnect

```
int mysql_pconnect (string [chaine_connexion],
                    string [nom],
                    string [motPasse])
```

Cette fonction est identique à `mysql_connect()`, mais elle ouvre une connexion *persistante* non refermée à la fin du script. En fait, pour un hôte et un nom d'utilisateur donnés, la fonction ne crée une connexion que lors du premier appel. Les appels suivants (avec les mêmes arguments) réutilisent la connexion qui existe toujours. L'intérêt est d'améliorer les performances en évitant de créer répétitivement des connexions.

mysql_ping

```
bool mysql_ping (int [connexion])
```

Teste la connexion avec MySQL, et se reconnecte si elle a été perdue.

mysql_query

```
int mysql_query (string requete,
                 int [connexion])
```

Exécute la requête (au sens large : toute commande MySQL) *requête*, en utilisant la connexion *connexion*. Si la requête échoue, par exemple à cause d'une erreur de syntaxe, la fonction renvoie `false`, sinon elle renvoie une valeur positive. Dans le cas de requêtes `SELECT`, la valeur renvoyée est l'identifiant du résultat qui peut être utilisé dans les fonctions *mysql_fecth_****.

mysql_real_escape_string

```
string mysql_real_escape_string() (string chaîne)
```

Effectue un échappement permettant d'obtenir une chaîne prête à l'insertion. Plus complète que `addSlashes()` car traite, outre les apostrophes, des données comme `NULL`, `\x00`, `\n`, `\r`, `\`, et `\x1a`. À utiliser donc pour insérer des données binaires dans une base.

mysql_result

```
int mysql_result (int résultat,
                  int noLigne,
                  int noAttribut)
```

Cette fonction permet d'accéder directement à l'attribut *noAttribut* de la ligne *noLigne* dans le résultat identifié par *résultat*. Elle est connue comme étant particulièrement lente à cause de la complexité de l'opération demandée. Il est recommandé d'utiliser plutôt la famille de fonctions *mysql_fetch_****.

mysql_select_db

```
int mysql_select_db (string nomBase,
                     int [connexion])
```

Fait de *nomBase* la base courante. Les requêtes exécutées par la suite s'effectueront sur *nomBase*.

mysql_tablename

```
int mysql_tablename (int résultat,
                     int indice)
```

Cette fonction utilise un identifiant de résultat fourni par `mysql_list_tables()` ou par `mysql_list_dbs()`, et permet d'accéder, selon le cas, à la liste des tables ou à la liste des bases de données. Le paramètre *indice* doit être compris entre 0 et la valeur ramenée par `mysql_num_rows()` qui donne le nombre de tables (ou de bases) à récupérer.

mysql_unbuffered_query

```
int mysql_unbuffered_query (string requête,
                            int [connexion])
```

Équivalent à `mysql_query()`, sauf en ce qui concerne la manière dont la requête est exécutée. La fonction `mysql_query()` charge tout le résultat en mémoire, tandis que `mysql_unbuffered_query()` se positionne sur la première ligne et attend les appels aux fonctions `mysql_fetch_***()` pour accéder successivement aux lignes suivantes. Elle demande donc beaucoup moins de mémoire, puisqu'une seule ligne à la fois est chargée. L'inconvénient est que `mysql_num_rows()` ne peut être utilisée sur le résultat.

Index général

A

abstraction 116–138, 185
accès disque *voir* mémoire externe
affectation 33, 41, 49, 139, 172, 423, 426–428, 430, 443
  d'un objet 172, 443
  dans un tableau 49
affichage multi-pages 109–114
agrégation 307–309, 311, 312, 413–415
Ajax 14
ajout d'un attribut *voir* ALTER TABLE, commande SQL
aléatoire 393, 482, 491
ancre HTML xvii, 4, 6, 7, 295, 303, 311, 391
anonyme (utilisateur MySQL) 447, 448
Apache 4, 21, 23, 24, 450, 508
apostrophes (gestion des) *voir aussi* échappement v, 29, 45, 68, 69, 73, 77, 130, 173, 224, 225, 235, 320, 321, 493, 494, 515
association 181, 185–196, 401
  binaire 188–190
  ternaire 190–192
authentification *voir aussi* session v, xxi, 101, 109, 279–286, 458, 459
auto incrémentation *voir aussi* séquences v, 72, 73, 94, 200, 234–236, 514

B

balises HTML
  `<a>` 6
  `<br/>` 22, 41, 75, 490
  `<form>` 9, 91, 158, 161
  `<img>` 97
  `<input>` 10, 91, 164
  `<option>` 12
  `<select>` 12
  `<table>` 14, 143, 144
  `<td>` 140
  `<textarea>` 13
  `<tr>` 14, 111, 140
  `<ul>` 304

C

C++ 116, 121, 420, 430, 437
cardinalités 186, 190, 191, 194, 195
casse (sensibilité à) 31, 234, 235, 420, 487, 488
chaînes de caractères *voir aussi* échappement, expressions régulières v
  de longueur fixe *voir* CHAR, commande SQL
  de longueur variable *voir* VARCHAR, commande SQL
  PHP 422
  SQL 395

classe
abstraite 129, 131, 132, 139, 169, 239, 379
constante de 139, 158, 171
constructeur 122, 123, 125, 132, 134, 135, 144, 148, 158, 171, 172, 178, 342, 343, 346, 441, 442
CSS 111
destructeur 122, 123, 342, 441, 442
extension d'une 127–128
héritage 127
parente 126, 128, 134, 135, 228, 442
partie privée 121, 132, 441
partie protégée 121, 132, 441
partie publique 121, 441
spécialisation 126–127
surcharge 127–129, 132, 176, 340, 442
clonage d'un objet 172
clé d'une table 107, 183–185, 187, 188, 193–197, 200, 280–283, 301, 303, 304, 312
codage des paramètres dans une URL 67, 77, 82, 83, 113, 492, 525
commentaires 28, 420, 452
concaténation 22, 40, 45, 105, 147, 390
conception d'une base de données 181–193, 195, 197
constructeur *voir* classe
contrôle *voir aussi* expressions régulières v
de l'existence d'une variable 87
de type 14, 70, 86
des données HTTP 67–72
convention de nommage xxi, 234, 235
cookie 17, 56, 99–101, 105, 107, 109, 280–282
copie d'un objet *voir* clonage
création
d'un index *voir* CREATE INDEX, commande SQL
d'un utilisateur *voir* GRANT, commande SQL
d'une base de données *voir* CREATE DATABASE, commande SQL
d'une table *voir* CREATE TABLE, commande SQL
CSS *voir* feuilles de style

D

dates
affichages des 304
et MySQL 234, 281, 304, 396, 464
et PHP 21, 304, 496
destructeur *voir* classe
disque dur *voir* mémoire externe
droits d'accès 15, 34, 279, 399, 447–451, 459, 474, 509, 510
création 451

E

email
envoi d'un 64
E/A *voir* schémas entité/association
échappement *voir aussi* *magic_quotes_gpc()*, fonction PHP v, 45, 68–70, 74, 77, 113, 130, 235, 248, 422, 423, 470, 515
email
envoi d'un 77
entités 181, 185–197, 401
XML 343
entrées/sorties *voir* mémoire externe
erreurs *voir aussi* exceptions v
dissimulation avec l'opérateur 40, 123
niveau de tolérance 223, 225, 421
pendant un transfert de fichier 92, 226
erreurs (traitement des) 57, 58, 120, 123, 124, 221–223, 225–227, 230–232, 421

error log (fichier d'erreurs) 225
exceptions 120, 124–126
    gestionnaire d' 232
    sous-classes 227
expressions régulières 86–90

F

fichier
    copie 94, 226
    extension 39
    ouverture 95
    transfert *voir* transfert de fichier
fichiers de configuration 26, 203, 448, 452–456
Firefox 7, 9, 15, 23
flush *voir* rechargement de MySQL
fonctions
    avec nombre variable de paramètres 160, 435
    paramètres *voir* passage des paramètres
    retour de plusieurs valeurs 62, 498
    récursives 70, 303
formulaires HTML 7–15, 17, 29, 35–37, 42, 43, 46–49, 52, 53, 152–167, 205, 274, 280, 284, 289, 295, 298, 301, 309
    avec choix multiples 47
    d'insertion et de mise à jour 78–86
forum de discussion 299–304

G

gestionnaire d'erreurs *voir* erreurs
gestionnaire d'exceptions *voir* exceptions
guillemets *voir* apostrophes

H

HTML 5–16, 20–24, 29, 35–37, 40–42, 46, 51–53, 55, 117, 152, 153, 159, 164, 167, 205, 299, 304, 391, 419
HTTP 3, 4, 6, 16, 17, 98, 280
héritage orienté-objet *voir* classe

I

identifiant
    d'une session 281, 282
    d'une table *voir* clé
identité d'objet 139, 443
`include_path`, directive PHP 507
inclusion d'un fichier 38, 60–61, 83
index
    création 206
    mise à jour 206
    recherche *voir* recherche par index
InnoDB 455, 467
Insérer un tableau dans une chaîne de caractères 105

J

Java 121
Javascript 14, 23
jointure 32, 206, 291–295, 399–406, 408, 468, 474
    auto-jointure 402
    externe 404–405
    naturelle 404

L

liens HTML *voir* ancres HTML
Linux 447, 457

M

`magic_quotes_gpc` 247
majuscules et minuscules *voir aussi* casse v, 25, 31, 137, 138, 234, 235, 321, 344, 389, 395, 420, 435, 464
`max_file_size` 12

méthodes magiques
__call() 384
__call() 272
__get() 272
__set() 272
modification d'une table *voir* ALTER TABLE, commande SQL
Modèle-Vue-Contrôleur (*pattern*) 208, 241–243, 245, 251, 260, 265, 267, 268, 273, 277, 278, 289, 296, 357–359, 361, 362, 364, 369, 374, 376, 380, 443
MVC *voir aussi* Modèle-Vue-Contrôleur (*pattern*) v
myisamchk, programme 467
mysql, programme 18, 25–31, 33, 34, 36, 37, 203, 205, 399, 454, 472
mysqladmin, programme 471
mysqld, programme 18, 19, 24, 26, 39, 399, 448, 450, 453, 456, 457
mysqldump, programme 18, 456, 457
mysqlimport, programme 18, 470
mysqlshow, programme 472
mémoire
centrale 23, 41, 420, 453, 513
externe 455
méthode statique 293
méthodes magiques 149, 271

N

navigateur 3–7, 10–12, 15–17, 21–24, 34, 36, 41, 42, 458, 487, 491, 492

O

opérateurs PHP 40, 41, 45, 111, 393, 395, 405, 427–430
arithmétiques 427
chaînes de caractères 428
de bits 429
de comparaison 430
logiques 429
opérateurs SQL 45, 394, 395, 405
ORACLE 19, 135, 233–236, 395, 457
orienté-objet *voir aussi* classe v, xvii, xxi, 37, 41, 55, 56, 61, 86, 115–117, 140, 255, 333, 334, 337, 339, 340, 343, 346–348, 419, 435, 440, 442
méthodes 118
objets 117–123
programmation 115–178, 440–443

P

parcours séquentiel *voir* recherche séquentielle
passage des paramètres
d'un objet 61
par adresse *voir aussi* références v, 61–64
par valeur 61–64
PDO 121, 129
php.ini, fichier de configuration 45, 90, 92, 94, 95, 109, 225, 282, 507
phpMyAdmin 34
phpMyAdmin 25, 34, 36, 169, 205, 457–459
PHP_SELF 22
portabilité multi SGBD 233–239
PostgreSQL 19, 60, 129, 135, 137, 138, 233–237, 457
prog.xslt, programme 322
programmation
avec fonctions 56
orientée-objet 116–167

R

recherche
par clé 206, 282, 291, 312
par index 206
par intervalle 289, 394
redirection vers une autre URL 98
renommage *voir* AS, commande SQL

répertoire
lecture d'un 95
requêtes avec négation 411–412
requêtes corrélées 409–411
requêtes imbriquées 406–412
root
MySQL 34, 35, 447, 448, 454, 459
RSS 317
réentrant (script) 67
références PHP 61, 62, 64, 70, 135, 139, 172, 173, 314, 435, 436, 443

S

Safari 15
SAX *voir* XML
schémas entité/association 181–193
script courant *voir* `PHP_SELF`
scripts SQL 27–28
séquence SQL 235, 236
session (gestion de) 16, 17, 22, 56, 98–109, 280–286, 458
`Set-Cookie` 17
SimpleXML 333–335
spécialisation *voir aussi* classe v, 126–138, 339–348
SQL ANSI 198, 200, 201, 387, 395–397, 404, 405, 414, 461–463, 467, 475
SQLite 135, 233, 235, 236, 239, 457
superglobales (variables) 22
suppression d'un attribut *voir* ALTER TABLE, commande SQL
surcharge orientée-objet *voir* classe

T

tableaux HTML 14, 152, 153, 290, 295
classe de production de tableaux 140–152
tableaux PHP 37, 41, 49, 52, 111, 160, 164, 298, 311–314, 423–425, 441, 442, 489, 490, 493, 497–500
affichage avec *print_r* 152
associatifs 58, 81, 86, 105, 111, 160, 164, 311, 313, 423–425, 497
indicés 49, 111, 423–425
templates voir aussi XSLT v, 251–267, 317
traitement des erreurs 222
transfert de fichier 8–10, 12, 15, 90–98, 159

U

URL xxi, 6, 7, 15, 46, 295

W

Windows 447, 448, 453, 457

X

XML
attributs 321
déclaration 320
et bases de données 323–332
et XHTML 320
forme sérialisée, forme hiérarchique 320
instructions de traitement 322
programmation DOM 332
programmation SAX 332–348
sections littérales 322
syntaxe de 318–322
utilisation de 319
éléments 321
XSLT 317, 322, 333, 348–354

Z

Zend, framework xvii, xxi, 208, 239, 260, 267, 278, 357, 358, 360, 362, 364–369, 371, 373, 375–379, 384

Index des fonctions PHP

__call(), 384
abs(), 315, **486**
addSlashes(), 45, 74, 235, **493**, 515
array(), **497**
array_key_exists(), **497**
arsort(), 311, 425, **498**
asort(), 311, 425, **498**
asXML(), **501**
attributes(), **501**
basename(), **486**
call_user_func(), 343, **486**
call_user_method(), **486**
ceil(), **486**
chdir(), **504**
checkdate(), **496**
children(), **501**
chop(), **493**
closedir(), 95, **504**
copy(), 94, **504**
count(), 52, 425, **498**
current(), 424, **498**
date(), 21, 283, **496**
define(), 78, **487**
defined(), **487**
dirname(), 486, **486**
each(), **498**
echo(), 21
empty(), 71, **486**
end(), **498**
ereg(), 89, 97, **487**
ereg_replace(), 89, 90, **488**
eregi(), **488**
eregi_replace(), **488**
error_log(), 226
error_reporting(), 223–225
EscapeShellCmd(), 487
eval(), 249, **487**
exec(), 487, **487**, 491
explode(), **493**
extension_loaded(), **488**
fclose(), **504**
feof(), 343, **505**
fgetc(), **505**
fgets(), **505**
file(), **505**
file_exists(), **505**
filesize(), 97, **505**
floor(), **487**
fopen(), 506, **506**
fpassthru(), **506**
fputs(), **506**
func_get_arg(), 435
func_get_args(), 435
func_num_args(), 435
get_browser(), 90
get_class(), 344
get_magic_quotes_gpc(), 69
getCode(), 227
getDate(), 234, 304
getdate(), **497**
getenv(), 22, **488**
getFile(), 227

getLine(), 227
getMessage(), 227
getType(), 426, **488**
Header(), 97, **488**
htmlEntities(), 76, 90, 265, **489**
htmlSpecialChars(), 71, 74, 77, 159, 174, 329
htmlspecialChars(), 74
implode(), **493**
in_array(), **499**
include(), 61, **506**
init_set(), 248
is_array(), 426, **489**
is_double(), 426, **489**
is_float(), 87, **489**
is_int(), **489**
is_long(), 426, **489**
is_numeric(), 87
is_object(), 426, **489**
is_string(), 87, 426, **489**
isSet(), 51, 71, 87, 421, **490**
key(), 424, **499**
krsort(), 311
ksort(), 311, 425, **499**
list(), **499**
ltrim(), **493**
mail(), 76, **490**
max(), **490**, **499**
md5(), 282, **490**
method_exists(), 343
min(), **490**, **499**
mkdir(), **506**
mkTime(), **497**
mysql_affected_rows(), **507**
mysql_change_user(), **508**
mysql_client_encoding(), **508**
mysql_close(), **508**
mysql_connect(), 37, 450, **509**
mysql_create_db(), **509**
mysql_data_seek(), **509**
mysql_db_name(), **509**
mysql_db_query(), **509**
mysql_drop_db(), **510**
mysql_errno(), **510**
mysql_error(), 37, 40, **510**
mysql_escape_string(), 45
mysql_fetch_array(), 510, **510**
mysql_fetch_assoc(), 80, **510**
mysql_fetch_field(), **511**
mysql_fetch_lengths(), **511**
mysql_fetch_object(), 37, 41, 234, 333, 511, **511**
mysql_fetch_row(), 37, 111, **511**
mysql_field_flags(), **512**
mysql_field_len(), **512**
mysql_field_name(), 111, **512**
mysql_field_seek(), **512**
mysql_field_table(), **512**
mysql_field_type(), **513**
mysql_free_result(), **513**
mysql_get_client_info(), **513**
mysql_get_host_info(), **513**
mysql_get_proto_info(), **513**
mysql_get_server_info(), **513**
mysql_info(), **513**
mysql_insert_id(), 94, **514**
mysql_list_dbs(), 509, **514**
mysql_list_fields(), **514**
mysql_list_tables(), **514**
mysql_num_fields(), 111, **514**
mysql_num_rows(), 113, 509, **515**, 516
mysql_pconnect(), 37, 39, 57, 58, **515**
mysql_ping(), **515**
mysql_query(), 37, 40, 110, **515**, 516
mysql_real_escape_string(), 74, 515, **515**
mysql_result(), **516**
mysql_select_db(), 37, 40, **516**
mysql_tablename(), **516**
mysql_unbuffered_query(), **516**
next(), 424, **499**
nl2br(), 75, **490**
oci_fetch_object(), 234
opendir(), 95, 504, 506, **506**
parse(), 343
passthru(), **491**

PASSWORD(), 473
pg_fetch_object(), 234
phpInfo(), **491**
phpinfo(), 488
pow(), 315
prev(), 424, **500**
print(), **491**
print_r(), 152, **491**
printf(), 21, **491**
putenv(), **491**
rand(), **491**
readdir(), 506, **506**
readfile(), 98, 506, **507**
readir(), 95
rename(), **507**
require(), 59, **507**
require_once(), 59, 247, 364
reset(), **500**
restore_error_handler(), 232
rmdir(), **507**
round(), **492**
rsort(), **500**
session_destroy(), 107, 287
session_id(), 107, 109
session_start(), 107, 109
set_error_handler(), 230
set_exception_handler(), 230
set_include_path(), 247
SetCookie(), 99, 100, 280, **492**
SimpleXML_import_DOM(), **501**
SimpleXML_load_file(), **500**
SimpleXML_load_string(), **500**
sort(), 425, **500**
sqrt(), 315
strchr(), **493**
strcmp(), **494**
strcspn(), **494**
strip_tags(), 71, 76, 77
stripSlashes(), 45, 69, 70, **494**
strlen(), **494**
strpos(), **494**
strrchr(), **495**
strrev(), **495**
strrpos(), **494**
strspn(), **495**
strToLower(), 137, **495**
strToUpper(), **495**
substr(), **495**
substr_count(), **494**
SYSDATE(), 481
system(), 491
time(), 492, **497**
trigger_error(), 226, 230
trim(), 338, **495**
unset(), 107, 441, **492**
urlDecode(), **492**
urlEncode(), **492**
urlEncode(), 67, 83, 113
xml_error_string(), **501**
xml_get_current_byte_index(), **501**
xml_get_current_column_number(), **502**
xml_get_current_line_number(), **502**
xml_get_error_code(), 501, **502**
xml_parse(), 335, 337, 343, **502**
xml_parse_into_struct(), **503**
xml_parser_create(), 335, **502**
xml_parser_free(), **502**
xml_parser_get_option(), **503**
xml_parser_set_option(), **503**
xml_set_character_data_handler(), 337, **503**
xml_set_default_handler(), **503**
xml_set_element_handler(), 337, **503**
xml_set_external_entity_ref_handler(), **503**
xml_set_notation_decl_handler(), **504**
xml_set_object(), 339, 343, **504**
xml_set_processing_instruction_handler(), **504**
XPath(), **501**

Index des commandes SQL

ABS(), **475**
ACOS(), **475**
ADD, 204
ADDDATE(), 476
ALL, 412
ALTER TABLE, 204, 465
    ADD, 204
    DROP, 205
    MODIFY, 204
AS, 292, 309, 391, 393
ASCII(), **475**
ASIN(), **475**
ATAN(), **475**
ATAN2(), **475**
AUTO_INCREMENT, 72, 73, 200, 474
BETWEEN, 45, 394
BIGINT, 462, 463
BIN(), **476**
BINARY, 464
BINARY VARCHAR, 395
BIT_COUNT(), **476**
BLOB, 395, 461, 462, 464, 467
CEILING(), **476**
CHAR, 198, 462–464
CHAR(), **475**
CHAR_LENGTH(), 480
CHARACTER_LENGTH(), 480
CHECK, 201, 202, 467
CONCAT(), **476**
CONSTRAINT, 467
CONV(), **476**
COS(), **476**
COT(), **476**
COUNT(), 307, 414
CREATE DATABASE, 26, 203, 466
CREATE INDEX, 475
CREATE TABLE, 25, 27, 35, 198, 466
CURDATE(), **476**
CURRENT_DATE(), 476
CURRENT_TIME(), 476
CURRENT_TIMESTAMP(), 481
CURTIME(), **476**
DATABASE(), **476**
DATE, 198, 396, 462, 464
DATE_ADD(), 390, **476**, 477
DATE_FORMAT(), 464, **477**, 478, 484
DATE_SUB(), **477**
DATETIME, 396, 462, 464
DAYNAME(), **477**
DAYOFMONTH(), **477**
DAYOFWEEK(), **478**
DAYOFYEAR(), **478**
DECIMAL, 198, 462, 463
DEFAULT, 198
DEGREES(), **478**
DELETE, 32, 33, 52, 417, 467, 507
DESC, 393
DESCRIBE, 27, 203, 472, 473
DISTINCT, 392, 408
DOUBLE, 462, 463
DOUBLE PRECISION, 462, 463
DROP, 204

DROP DATABASE, 466
DROP INDEX, 475
DROP TABLE, 27, 204
ELT(), **478**
ENCRYPT(), **478**
ENUM, 47, 462, 465
EXCEPT, 405, 406
EXPLAIN, 472
FIELD(), **478**
FIND_IN_SET(), **478**
FLOAT, 462, 463
FLOOR(), **478**
FLUSH, 471
FLUSH PRIVILEGES, 449
FOREIGN KEY, 200–202, 467
FORMAT(), **478**
FROM, 31, 291, 292, 388–391, 393, 399–401, 412, 450, 468, 469
FROM_DAYS(), **478**
FROM_UNIXTIME(), **478**
GET_LOCK(), **479**, 482
GRANT, 26, 449–451, 473, 474
GREATEST(), **479**
GROUP BY, 308, 309, 413, 414
HAVING, 415
HEX(), **479**
HOUR(), **479**
IF(), **479**
IF EXISTS, 466
IFNULL(), **479**
IGNORE, 466
IN, 394, 406
INSERT, 28, 29, 31, 35, 36, 52, 94, 235, 298, 416, 449, 457, 464, 469–471, 507, 514
INSERT(), **479**
INSTR(), **479**, 480
INTEGER, 25, 198, 204, 462, 463
INTERSECT, 405, 406
INTERVAL(), **479**
INTO OUTFILE, 399, 469
IS NOT NULL, 397
IS NULL, 397
ISNULL(), **479**
JOIN, 469
KILL, 472
LAST_INSERT_ID(), 73, 474, **479**
LCASE(), **480**
LEAST(), **480**
LEFT(), **480**
LEFT OUTER JOIN, 404
LENGTH(), **480**
LIKE, 45, 290, 395, 472, 473
LIMIT, 398
LOAD DATA, 29, 30, 399, 469, 470
LOCAL, 30
LOCATE(), **480**
LOCK, 473
LOCK TABLES, 473
LOG(), **480**
LOG10(), **480**
LONG, 464
LONGBLOB, 462
LONGTEXT, 462
LPAD(), **480**
LTRIM(), **480**
MAX(), 414
MEDIUM, 464
MEDIUMBLOB, 462
MEDIUMINT, 462, 463
MEDIUMTEXT, 462
MID(), **480**, 483
MIN(), 412
MINUTE(), **480**
MOD(), **481**
MODIFY, 204
MONTH(), **481**
MONTHNAME(), **481**
NOT, 394
NOT NULL, 308, 398, 413, 469
NOT NULL, 198–200, 204
NOW(), **481**
NULL, 308, 391, 396–398, 404, 413, 416, 469, 470, 482
NUMERIC, 462, 463
OCT(), **481**

OPTIMIZE TABLE, 467
OR, 394, 397
ORDER BY, 393, 469
PASSWORD(), 451, **481**
PERIOD_ADD(), **481**
PERIOD_DIFF(), **481**
PI(), **481**
POW(), **481**
POWER(), 481
PRIMARY KEY, 199, 200
QUARTER(), **481**
RADIANS(), **482**
RAND(), 393, **482**
REAL, 462, 463
RELEASE_LOCK(), **482**
RENAME, 204
REPEAT(), **482**
REPLACE, 469–471
REPLACE(), **482**
REVERSE(), **482**
REVOKE, 449, 451, 474
RIGHT(), **482**
ROUND(), **482**
RPAD(), **482**
RTRIM(), **482**
SEC_TO_TIME(), **483**
SECOND(), **483**
SELECT, 31–33, 35, 235, 291, 292, 309, 388, 391–393, 399, 401, 468, 469, 471, 472, 474, 510, 511, 514, 515
SESSION_USER(), 484
SET, 33, 47, 462, 465, 473
SHOW, 472
SIGN(), **483**
SIN(), **483**
SMALLINT, 462, 463
SOUNDEX(), **483**
SPACE(), **483**
SQRT(), **483**
STRCMP(), **483**
SUBSTRING(), 480, **483**
SUBSTRING_INDEX(), **483**
SYSDATE(), 390
SYSTEM_USER(), 484
TAN(), **483**
TEMPORARY, 467
TEXT, 47, 198, 395, 462, 464, 467
TIME, 198, 462, 464
TIME_FORMAT(), **484**
TIME_TO_SECOND(), **484**
TIMESTAMP, 462, 464
TINY, 464
TINYBLOB, 462
TINYINT, 462, 463
TINYTEXT, 462
TO_DAYS(), **484**
TRIM(), **484**
TRUNCATE(), **484**
UCASE(), **484**
UNION, 405
UNIQUE, 200, 206
UNIX_TIMESTAMP(), **484**
UNKNOWN, 396, 397
UNLOCK TABLES, 473
UNSIGNED, 463
UPDATE, 32, 33, 52, 298, 417, 451, 464, 471, 507
USAGE, 450, 451
USE, 27, 37, 450, 455, 471
USER(), **484**
VALUES, 29, 416
VARCHAR, 25, 198, 204, 395, 462–464
VERSION(), **484**
WEEK(), **484**
WHERE, 31–33, 201, 291, 292, 307, 309, 388, 389, 393, 397, 399, 401, 403, 406, 415, 417, 455, 467, 468, 471
YEAR, 462, 464
YEAR(), 390, **484**
ZEROFILL, 462, 463

Table des figures

1 Barre d'outils Web Developer xix

1.1 Architecture web 4
1.2 Architecture basique d'une application web 8
1.3 Présentation d'un formulaire avec Firefox 9
1.4 Serveur et clients de MySQL. 18
1.5 Architecture d'un site web avec MySQL/PHP 24
1.6 Page d'accueil de phpMyAdmin 34
1.7 Actions sur une base avec phpMyAdmin 35

2.1 Le schéma de l'application d'envoi d'un e-mail 65
2.2 Formulaire d'envoi d'un e-mail 66
2.3 Affichage du texte d'un e-mail comprenant des balises 75
2.4 Formulaire en modification du film *Vertigo* 81
2.5 Page de mise à jour des films 83
2.6 Le formulaire, au début de la session 103
2.7 Après choix du plat et de l'entrée 106
2.8 Le menu est choisi 106
2.9 Le formulaire d'interrogation, avec affichage multi-pages 114

3.1 Application avec objets. 118
3.2 Gestion des exceptions. 124
3.3 Affichage des deux tableaux. 146
3.4 Conception de la classe `Formulaire` 153
3.5 Affichage du formulaire de démonstration. 156
3.6 Organisation de la classe `Formulaire` 157

3.7 Affichage de l'interface sur la table *Carte*. 170

4.1 Schéma de la base de données *Films* 185
4.2 Représentation des entités 188
4.3 Association entre deux ensembles. 189
4.4 Représentation de l'association. 189
4.5 Association `(Acteur,Film)` 190
4.6 Associations entre *Artiste* et *Film*. 190
4.7 Association ternaire. 191
4.8 Graphe d'une association ternaire. 191
4.9 Transformation d'une association ternaire en entité. 192

5.1 Environnement de développement Eclipse pour PHP 209
5.2 Configuration de la connexion au serveur CVS 211
5.3 Exploration du répertoire distant CVS 211
5.4 Validation de modifications, et transfert sur le serveur CVS 212
5.5 Exemple de page produite par PhpDoc 213

6.1 Aperçu général d'une application MVC 242
6.2 Organisation du code 244
6.3 Tout le code HTML est produit avec PHP. 252
6.4 Affichage du document résultat 255
6.5 Template contenant une liste 256
6.6 Page de recherche des films 262
6.7 Le résultat d'une recherche 262
6.8 Formulaire d'inscription sur le site 273

7.1 Page d'accueil après identification d'un internaute 288
7.2 Formulaire de recherche des films 289
7.3 Formulaire de notation des films 295
7.4 Présentation d'un film 301
7.5 Formulaire d'accès aux prédictions 310
7.6 Liste des films les plus appréciés 312

8.1 Import et export de données XML dans le WEBSCOPE 318
8.2 Résultat de la transformation XSLT 354

9.1 Page d'accueil du ZSCOPE 361
9.2 Organisation (minimale) du code pour une application Zend . . . 361

10.1 Un échantillon de la base *Films* 388
10.2 Table *FilmXArtiste*, définie par la clause `FROM Film, Artiste`. . 400

INFOPRO

TYPE D'OUVRAGE

L'ESSENTIEL	SE FORMER	RETOURS D'EXPÉRIENCE

MANAGEMENT DES SYSTÈMES D'INFORMATION

APPLICATIONS MÉTIERS

ÉTUDES, DÉVELOPPEMENT, INTÉGRATION

EXPLOITATION ET ADMINISTRATION

RÉSEAUX & TÉLÉCOMS

Philippe Rigaux

PRATIQUE DE MySQL ET PHP

Conception et réalisation de sites web dynamiques

4e édition

Ce livre s'adresse à tous les développeurs web, aux étudiants en informatique et à tous ceux qui veulent s'initier à la pratique de ces deux piliers du monde Open Source que sont MySQL et PHP.

Le principal objectif de ce livre est d'exposer de manière claire et précise les techniques utilisées pour la création de sites web interactifs avec MySQL et PHP.

Plutôt que de donner des recettes sans justification il cherche à expliquer au lecteur *pourquoi* et *comment* recourir à telle technique plutôt qu'à telle autre. Il lui permet ainsi de mieux assimiler ses connaissances et d'être capable de les réutiliser dans d'autres contextes.

Un site web complète ce livre avec des exemples, des liens, des compléments utiles et tout le code de l'application consacrée au cinéma qui sert d'exemple tout au long du livre.

Les trois premières éditions de cet ouvrage, qui est rapidement devenu le titre de référence en français sur MySQL et PHP, avaient été publiées par O'Reilly France.

Cette quatrième édition s'enrichit d'un nouveau chapitre sur les environnements de programmation PHP/MySQL et introduit notamment le *Zend Framework*.

PHILIPPE RIGAUX

est professeur des universités à Paris-Dauphine. Il enseigne les bases de données ainsi que les techniques de programmation et les langages web. Il intervient également comme consultant en entreprises et est l'auteur de cinq livres en français et en anglais et de plus de cinquante publications internationales.

SUPPLÉMENTS en ligne

ISBN 978-2-10-053752-5

www.dunod.com

DUNOD